Journal of Visual Languages and Computing 21 (2010) 171–183

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

Mental imagery and software visualization in high-performance software development teams

Marian Petre *

Centre for Research in Computing, Open University, Milton Keynes MK7 6AA, UK

ARTICLE INFO

Article history:
Received 28 September 2006
Received in revised form
16 October 2009
Accepted 9 November 2009

Keywords:
Software visualization
Empirical studies
High-performance programming
Teamwork

ABSTRACT

This paper considers the relationship between mental imagery and software visualization in professional, high-performance software development. It presents overviews of four empirical studies of professional software developers in high-performing teams: (1) expert programmers' mental imagery, (2) how experts externalize their mental imagery as part of teamwork, (3) experts' use of commercially available visualization software, and (4) what tools experts build themselves, how they use the tools they build for themselves, and why they build tools for themselves. Through this series of studies, the paper provides insight into a relationship between how experts reason about and imagine solutions, and their use of and requirements for external representations and software visualization. In particular, it provides insight into how experts use visualization in reasoning about software design, and how their requirements for the support of design tasks differ from those for the support of other software development tasks. The paper draws on theory from other disciplines to explicate issues in this area, and it discusses implications for future work in this field.

1. Introduction: the relationship between software visualization and human tasks

Richard Hamming wrote that *"The purpose of computing is insight, not numbers"* [1]. Arguably, the role of software visualization lies in supporting and promoting insight about software—and hence supporting human reasoning, both individual and shared. In order to do so, software visualization must be consistent with human perception and cognition, as well as with human tasks that frame them. Hence, it is difficult to consider software visualization without also considering how effective software developers reason about software, and how that human capability might be supported and extended using visual cues and devices [2]. This paper considers the relationship between reasoning about software (including mental imagery employed during design and communicating design ideas) and software visualization in professional, high-performance software development.

Software visualization has its roots in the earliest software development practice, when programmers watched the lights on the computer's control panel and listened to the sounds of disk access to try to understand what the program was doing in the absence of other perceivable cues. Software lacks tangibility and visibility (e.g., What does a compiler look like? What is the size, weight and shape of an operating system?). Code may be manifest, but how code *works* must be discovered and understood. Software visualization is concerned with using visual or graphical techniques (sometimes enhanced by audio) to provide perceivable cues to potentially obscure aspects of software systems, in order to reveal patterns and behaviours that inform software comprehension through all stages of software development.

* Tel.: +44 1908 65 33 73; fax: +44 1908 65 21 40.
E-mail address: m.petre@open.ac.uk

1045-926X/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2009.11.001

This paper examines the relationship between software visualization and effective software developers' reasoning specifically *during design and generation*. Goel [3] argues, in the context of external representation, that there is a principled distinction to be made between design and non-design problems. That distinction is pertinent here. There is substantial take-up of software visualization in the comprehension and maintenance of legacy systems, for example. Yet it is not clear that visualizations designed to support comprehension and debugging are effective in supporting design and initial development. It is difficult to consider software visualization without also considering the task it is meant to support, and it is unlikely that any single software visualization tool can address all software development tasks simultaneously [4]. Each task requires a different understanding – and a different view – of the software. That view must be expressed through visual cues and devices that bridge effectively between properties of software and human reasoning. The challenge is to identify the most appropriate visualization for a given task.

This paper focuses on experts and their high-performing teams, on the basis that they have a track record of effective reasoning and effective communication about software. The paper looks across a series of empirical studies of expert and high-performing team behaviour and reasoning to consider whether there are lessons from the experts' own imagery and tools that might inform software visualization to support software design and generation. Each study addresses a different facet of the continuum between what experts conceive in their minds and what they and their teams create in code. Hence, in describing each study in turn, the paper steps from expert mental imagery, through externalization of that imagery in communicating design concepts, to visualization tools experts use (or do not use) and what characterizes them. It considers which tasks experts choose to support with visualizations during software design and generation, and the nature of the visualizations they use to address them. The purpose is to articulate what expert software developers visualize in their minds and how they represent their conceptions externally, in order to consider what roles visualization might play in supporting their reasoning and communication during development tasks.

The sections are organized as follows: a distillation of relevant findings from the literatures on mental imagery, expert problem-solving, memory and schema; a description of the context common to all of the studies; descriptions of each of the empirical studies; a discussion of the implications of the findings for software visualization; and a conclusion.

2. Background: relevant observations from the literature

Software visualization has developed apace with visual techniques, from early 'pretty printers', which use typographic enhancements such as indentation and colour coding (e.g., [5]), to 3D 'landscapes' representing the structure of large software systems, to multiple, linked, dynamic visualizations of the interaction of system components at runtime. The literature provides a number of good overviews (e.g., [6–9]) and taxonomies (notably, [10,4]). Various visualization tools are available on the web, on individual websites, and in compilations such as 'SCG Smallwiki CodeCrawler: a non-exhaustive list of software visualization tools' [11].

Yet, despite developments in technology and techniques, questions remain about how well visualizations actually support human reasoning and tasks—and about how well they relate to the mental imagery that proficient software developers use in reasoning and communicating about software. There is widespread anecdotal evidence (e.g., Lammers's interviews of well-known programmers [12]) that programmers make use of visual mental images and mental simulations when they are designing programs. There are broad, well-established literatures on mental imagery, expert problem-solving, memory and schema that, although they do not specifically address software design, can contribute to our thinking about imagery and representation in this context. Although there is apparently little empirical research on programmers' mental imagery *per se*, there is a well-established literature on expert problem-solving, which, taken as a body, suggests a pattern of mental structure building preliminary to efficient externalization of solutions.

Lesson 1: Visualizations should provide appropriate *abstractions*, while supporting detailed enquiry and systematic exploration—the implication is that visualizations should be explorable and manipulable at different levels.

Rationale: Expert problem solvers (across domains) differ from novices in both their breadth and organization of knowledge; experts store information in larger chunks organized in terms of underlying abstractions (see [13,14] for reviews). When categorizing problems, experts sort in terms of underlying principles or abstract features (whereas novices tend to rely on surface features) (e.g., [15,16]). Experts form detailed conceptual models incorporating abstract entities rather than concrete objects specific to the problem statement [17]. Their models accommodate multiple levels and are rich enough to support mental simulations [18,19]. There is a recurrent theme of overview and abstraction in expert reasoning, which has relevance for tool-building, for which it raises the issue: how can a tool show the essence rather than the superficial?

Lesson 2: Visualizations should offer *relevance*: the visualization should map to appropriate referents and schema.

Rationale: It was Bartlett [20] who first suggested that memory takes the form of schema, which provide a mental framework for understanding and remembering. In general, the term 'schema' is used to indicate a form of mental template that organizes cognitive activity such as memory, reasoning, or behaviour. Key characteristics or elements in the schema have 'slots' in the mental template: the slots represent the range of values acceptable for those key characteristics. Schema may be of varying levels of complexity and abstraction; their

importance is in providing structure and economy. Chi et al. [21] suggest that the nature of expertise is due largely to the possession of schemas that guide perception and problem solving—i.e., experts have more and better schemas than novices. Simon [22] observes that, when a task is ill-defined, users resort to pre-existing concepts: stereotypes, schemata, or other knowledge. Cole and Kuhlthau [23] see the use of schemata as fundamental to sense-making at the outset of problem solving: the problem-solver invokes a schema or model of the problem in order to create a frame of reference and hence to identify the initial problem state. On the one hand, the use of existing schemata enables the user to take some action in unfamiliar or ill-defined tasks. On the other hand, the use of existing schemata can lead to misconception, misaction, or fixedness [24].

Lesson 3: Visualizations should support *selection of focus*: allowing selection of focus of particular subsets of information (while maintaining access to other information), allowing selective highlighting of different aspects of the software, matching representation to focus, supporting different perspectives through different visualizations.

Rationale: Experts often engage in systematic exploration, whereas novices are less likely to engage in exploratory interactions [25].

Lesson 4: Visualizations should *associate perceptual cues with aspects of semantic importance*—for example, associating vividness and saliency with key functionality (and avoiding drawing attention to what is less important). Visualizations should provide perceptual cues for deep structures and functionality—in order to assist in the detection of underlying patterns and structures.

Rationale: The well-established literature on memory (dating back over a century to pioneers such as Ebbinghaus [26]) presents a number of findings that highlight the role of perceptual cueing in recall. Things that are easily visualized can be held in memory more easily—there is an entire literature on mnemonics, for instance (e.g., [27]). More vivid images tend to be better remembered, although there is little or no correlation between the vividness and the accuracy of a memory [28]. Order of presentation affects recall: the last items in a list tend to be remembered best ('recency'), as do those that are presented earliest and are therefore most rehearsed or processed (primacy). Recall is weaker than recognition. Memory is an active encoding process, not a passive recording process, and is subject to distortions and bias at the encoding and retrieval stages. The encoding typically involves schemata, and this has many implications for the field of visualization research (Bartlett [20] and others).

Lesson 5: Visualizations should *use multiple, varied perceptual cues*, and they should *link different visualizations* or representations to promote discovery and insight.

Rationale: The psychology literature on mental imagery, informed recently by neurological and experimental evidence that imagery involves different systems (visual, spatial, verbal, temporal, lessonal/semantic), which are usually handled in different parts of the brain, gives reason to consider that we maintain multiple mental representations and that imagery is encoded in multiple, different modalities and systems (e.g., [29–31]). Many of the hypotheses about sources of insight are based on interactions between encodings. For example, Anderson and Helstrup [32] argue that mental imagery is a source of discovery and synthesis. Bartlett [33] wrote that imagery leads into bypaths of discovery. Logie [34] described an economy of images in memory, through which access to previously unrelated bits of information might be achieved: many informative elements are integrated together in a structural whole, increasing the available amount of information in working memory. Lindsay [35] claimed that images allow inferences that are not based on proof procedures. The most pertinent shortcoming of the imagery literature is the tasks involved. Most of the studies deal with particular, usually concrete or real-world images and simple tasks. Hence their conclusions might not generalize to a realm in which the imagery concerns complex, abstract, imagined images.

Lesson 6: Visualizations should provide *support for early conceptual activity*.

Rationale: Experts tend to spend more time than novices planning and evaluating. Experts are better able to form overviews, but thereafter they take longer to develop their understanding and representations, and they consider interactions among functions or components of a system more fully [36].

Similarly, Card et al. [37] "propose six major ways in which visualization can amplify cognition…: (1) by increasing the memory and processing resources available to the users, (2) by reducing the search for information, (3) by using visual representations to enhance the detection of patterns, (4) by enabling perceptual inference operations, (5) by using perceptual attention mechanisms for monitoring, and (6) by encoding information in a manipulable medium." (p. 16) Do we really understand enough about the design of software visualizations to realize these gains? Petre et al. [2] raised a number of cognitive issues facing software visualization, many of them identifying the gaps between the sorts of potential gains enumerated above, and the specific insights and techniques required to provide sufficiently selective and well-designed visualizations, which focus appropriately on human cognitive activity. Again, the missing link is between human reasoning about software and what is visualized. The goal of the series of studies presented in this paper was to examine how expert software developers realize the link between their own reasoning and their representations.

3. Context: which experts, which software, which overall tasks?

To reiterate: the focus of this paper is on the relationship between expert mental imagery and software visualization in the context of *software design and generation*, rather than legacy software comprehension. The motivation arises from a broader interest in what distinguishes expert reasoning and behaviour in the development of software, and how they might be supported, and the goal of this series of studies was to

expose how experts bridge between their internal reasoning and various externalizations, realizations, and implementations of that reasoning. From this, it was hoped that lessons could be identified that would benefit software visualization. Hence the studies examined how experts reasoned about software, elicited their imagery, and examined the sorts of tools they used and built to support their own reasoning.

All four studies fit into the broad category of research, which Ball and Ormerod [38] characterized as 'cognitive ethnography'. They view expert activity in context, while contributing to the understanding of cognition 'in-the-head', hence attending to "the interplay between people-laden contexts and expert cognition" (p. 148). Ball and Ormerod characterize cognitive ethnography as follows:

observationally specific: using small-scale data collection based around representative time slices of situated activity.

purposive: focusing on selected issues within existing work practices, and

verifiable: in terms of validating observations across observers, data sets and methodologies.

The studies reported here involved observing and characterizing the nature of practice as it is found. The endeavour concerns identifying key concepts, representations, strategies, and processes in expert practice, rather than profiling cases in terms of existing categories or theories, and therefore it is elicitative, descriptive, and inductive in nature, and a qualitative approach was adopted as appropriate. It is the existence and nature of the expert practice that is of interest, rather than its frequency. Hence, the emphasis is on qualitative rather than quantitative analysis, although a quantitative consideration of prevalence and impact might well be a matter for further work.

The approach provides a means for identifying patterns across individuals: identifying phenomena of interest, cataloguing behaviours and strategies, identifying key factors, and focusing questions for further study. It is informed by (and triangulates among) a variety of inputs, including direct observation, talk-aloud protocols, interviews, environments and artefacts.

3.1. The experts

The experts, from both industry and academia, and from several countries in Europe and North America, share the same general background: all have ten or more years of programming experience; all have experience with large-scale, real-world, real-time, data- and computation-intensive problems; and all are acknowledged by their peers as expert. All are proficient with programming languages in more than one paradigm. The coding language used was not of particular interest in these investigations, but, for the record, a variety of styles was exercised in the examples, using languages including APL, C, C++, Hypercard, Java, common LISP, macro-assembler, Miranda, Prolog, and SQL. Their preferred language was typically C or C++, because of the control it afforded, but the preference did not exclude routine verbal abuse of the language.

3.2. The companies and teams

All were small teams of 3–12 members, all included at least one expert software developer of the calibre of 'super designer' [39], and all were in companies where the generation of intellectual property and the anticipation of new markets characterized the company's commercial success. All were high-performance teams: effective intellectual-property-producing teams that tend to produce appropriate products on time, on budget, and running the first time. The companies were small, not more than 200–300 employees, although some were autonomous subsidiaries of much larger companies.

3.3. The domains

Most teams were undertaking large, long-term (1- to 2-year) projects. Often the software was one component of a multi-disciplinary project including computer hardware and other technology. Industries included computer systems, engineering consultancy, professional audio and video, graphics, embedded systems, satellite and aerospace—as well as insurance and telecommunications. Software developers generate between 5 and 10,000 lines of code per compile unit, typically around 200 lines per compile unit, with on the order of 3000 files per major project.

It is important to note that these experts work in relatively small companies or groups that typically produce their own software rather than working with legacy systems. The software they produce is 'engineering software' rather than, for example, information systems, although products may include massive data handling and database elements.

3.4. Limitations

It is important to note that this work is based on studies in a specific context, one determined pragmatically—by which companies were willing to allow access to their expert software developers. The results presented may not generalize beyond this variety of design and this style of working.

Experts are well-known for rationalizing their practice 'on-the-fly'. As reported by Schooler et al. [40], there is evidence that solving insight problems relies on essentially non-reportable processes, even that verbalisation interferes with some important thought processes. The dangers of elicitation techniques such as interviews are well-known, as reported in [41]. The main problems with data from verbal reports are the likelihood that cognitive processes of interest are not accessible to introspection, and the possibility that the experimenter might bias the response by asking only for certain types of report. On the other hand, although subjective tests may be suspect, they have in some cases been shown to be reliably consistent, and to produce results just as good as those from more

objective tests [42]. There is some evidence that self-ratings do correlate with demonstrated ability [43] and are stable in cases where they do. These studies relied on subjects whose reports of activity in earlier studies corresponded well to other evidence of their activity, such as notes and observed actions, i.e., it relied on subjects who appeared to be 'good self-reporters'.

Similarly, the quality of observation depends on the quality of the observer. Rigour in qualitative analysis derives from systematic practice, including the 'operationalisation' of categories as categorization rules and exemplars that can be employed by other researchers; coding and indexing in a way that ensures an audit trail between data and conclusions, and vigilance in seeking counter-examples to and inconsistencies with apparent patterns. The limitations of verbal reports highlight the need for different forms of input, and for 'triangulation' among inputs in order to examine a proposition. Validation in these studies is based on critical scrutiny of the findings by the informants, and on subsequent expert review.

4. Study 1: Software developers' mental imagery

So what evidence is there about the nature of software developers' mental imagery? A previous paper [44] describes a study into the mental imagery of ten individual expert software developers, who were questioned directly regarding the nature of their mental representations while they were engaged in a design task.

Study description—observation and interview of 10 software developers: Ten expert software developers were asked to design solutions to one of four problems (an interactive noughts and crosses player, an academic timetable maker, a sub-anagram solver, or a pinball path predictor) or to a problem of their choice. The experts were asked to imagine themselves free of coding restrictions, and they were not asked to implement the solutions as code. The small but non-trivial problems were chosen to evoke rich discussions by addressing classic issues in data representation and by admitting both standard and innovative treatments (the effective task is therefore the generation of potential solutions within this framework). Transcripts taken throughout the task provided a record of software developer remarks and the contexts in which they were made; i.e., the transcripts provided a record of which remarks were spontaneous and which were prompted by questions keyed to the moments when (or the moments just after) the experts showed signs of internally focussed thinking, such as pauses in writing activity, closing eyes, staring at blank paper (fixedly or with eye movement), or gesturing in the air. Prompting questions were general, e.g.,: "What do you see?", "What can you hear?", "What's going on?", "Where are you now?". After the tasks were completed, the experts were interviewed about their previous responses and about their imagery in general. All notes and other products were collected, and the sessions were recorded.

The analysis was inductive and iterative. Notes and transcripts were examined for the imagery descriptions they contained, and those imagery accounts were grouped in terms of patterns and common elements. The groups were data-driven, not shaped by any pre-existing framework. Attention was given to the context in which the account was given. Particular attention was given to the spontaneity of the accounts (as reflected by the fluency and immediacy of description and by the amount of prompting required), to the programmer's satisfaction with the account, and to any discrepancies. Patterns (and observations about them) were articulated explicitly. The data were re-examined, seeking contradictions or inconsistencies that might challenge the induced patterns. Hence, questions or conjectures that arose during examination of the data became the focus of another systematic iteration through the data. This is consistent with the 'constant comparison' method suggested by Glaser and Strauss [45], although this study did not include the iterations of data collection suggested by grounded theory.

This study consisted of structured observations and interviews attempting to elicit introspective reports of mental imagery, *not* a controlled laboratory experiment. The experts, all familiar informants whose reports of activity in earlier studies corresponded well to other evidence of their activity, demonstrated a readiness to describe the form and content of their thinking. The main images are as follows, each with an example from one of the informants in italics (see [44] for a more complete summary):

dancing symbols ("text with animation"): "… it moves in my head … like dancing symbols … I can see the strings [of symbols]… assemble and transform, like luminous characters suspended behind my eyelids …"

mental descriptions or discussion (mental verbalisations): *"I'm just talking to myself …"*

auditory images (auditory presentations of solution characteristics, with auditory qualities like loudness or tone reflecting some aspect of the solution): *"It buzzes … there are things I know by the sounds, by the textures of sound or the loudness … it's like I hear the glitches, or I hear the bits that aren't worked out yet …"*

visual imagery, e.g.: "values as graphs in the head … flip into a different domain … transform into a combined graph … (value against time; amplitude against frequency; amplitude against time) …"

machines in their minds (dynamic mental simulations, of three sorts: abstract machines, pictures of implementations, and mechanical analogies), e.g.: "… *nets of stuff … poking inputs and see what filters through … sucking: I'm ready, send me another one …"*

surfaces (a strongly spatial, mathematically oriented imagery of 'solution surfaces'), e.g.: "It's like driving across a desert looking for a well. What you actually have is good solutions distributed across this desert like small deep wells and your optimizer trundling along looking for them..."

landscapes (a strongly spatial imagery, a surface or landscape of solution components over which they could 'fly'), e.g.: "… it's like flying over landscape of stuff I'm thinking about, parts of the solution … I can see what the terrain is like, where I am and keep an

eye on stuff on the horizon, or I can close in on something…"

presences (a sort of imagery that was not verbal, visual, or physical; an imagery of presence (or knowledge) and relationship), e.g.: "… *no place holders, no pictures, no representation … just the notion, the symbol entities, semantic entities and the linguistic token … atomic notions. They just 'are'.*"

There were some common characteristics of the imagery, *viz:*

stoppably dynamic: All of the images were described as dynamic, but subject to control, so that the rate could be varied, or the image could be frozen.

variable selection: The 'resolution' of the imagery was not uniform; the experts chose what to bring into and out of focus.

provisionality: All of the imagery could accommodate incompleteness and provisionality, which were usually signalled in the imagery in some way, e.g., absence, fuzziness, shading, distance, change of tone.

many dimensions: All of the experts reported using more than four dimensions. The extra dimensions were usually associated with additional information, different views, or strategic alternatives.

multiplicity: All of the experts described simultaneous, multiple imagery. Some alternatives existed as different regions, some as overlaid or superimposed images, some as different, unconnected mental planes.

naming: Although some of the imagery was largely non-verbal, the experts all talked about the ready ability to label entities in the imagery.

The findings were presented to and discussed with the informants, who were asked to identify any inaccuracies. Although not all of the informants experienced all the forms of imagery collected, all were satisfied with the capture and categorization of their own.

5. Study 2: Externalization of mental imagery

A key question in this area is whether personal mental imagery *ever* becomes public or externalized. A follow-on question is whether personal mental imagery would be of any use if it does become public. Some images and imagery, for instance, may be extremely useful to the individual, but by their nature may be very difficult to describe verbally and to use as a shared metaphor, because they are not well suited to reification and shared physical representations (such as diagrams, gestures, physical analogies, etc.).

It seems intuitively obvious that there are times when imagery *does* become externalized and when the externalization is useful. Yet there is little published evidence of effective, direct externalization of personal mental imagery in software development, apart from introspective justifications for software tool design. This section reports a form of externalization that has been observed to occur naturally in high-performance development teams: when an individual's mental image becomes focal to team design activity and reasoning [46].

Study description—observation over time, focused on 5 teams in 3 companies, 10 projects overall: The evidence discussed here is a 'by-product' of other studies: initially, of the mental imagery study summarized above; subsequently, of a number of other *in situ* observational studies of early design activity. Those studies had other issues as their focus, for example design representations and processes used by multi-disciplinary concurrent engineering teams, representations (including ephemeral ones) used in early ideas capture, and group discussions and processes in very early conceptual design. Thus, the core evidence was accumulated from five different software development teams and ten different projects in three different companies over a period of some five years. The data collection was opportunistic, recording relevant examples as they arose naturally. The records collected for this study included audio recordings, detailed contemporaneous field notes, photographs of whiteboards and other artefacts, and photographs or photocopies of documents, including ephemera.

Again, the analysis was inductive, identifying patterns and common elements among the examples, following the iterative, data-driven process described for the previous study. Again, the captured accounts were presented to the informants, who were asked to scrutinize critically the accuracy of the records and their analysis. Only one example of a focal image is given below, but each group manifested at least one example, and the observations reported are representative of all examples.

One typical example arose in the context of the mental imagery study described above. The expert was thinking about a problem from his own work and articulated an image: "...the way I've organized the fields, the data forms a barrier between two sets of functions...It's kind of like the data forming a wall between them. The concept that I'm visualizing is you buy special things that go through a wall, little ways of conducting electrical signals from one side of a wall to another, and you put all your dirty equipment on one side of a wall full of these connectors, and on the other side you have your potentially explosive atmosphere. You can sort of colour these areas...there's a natural progression of the colours. This reinforces the position queues...There's all sorts of other really complex data interlinkings that stop awful things happening, but they're just infinitely complex knitting in the data. (Of course it's not pure data...most of the stuff called data is functions that access that data.) The other key thing...is this temporal business we're relying on...the program is a single-threaded program that we restrict to only operate on the left or on the right...a hah!...the point is that the connections to the data are only on one side or the other. The way I organize the data is...a vertical structure, and the interlinkings between data are vertical things...vertical interlinkings between the data tell me the consistency between the data, so I might end up, say, drawing between the vertically stacked data little operator diagrams..." After he described the image fully, he excused himself and went down the corridor to another team member, to whom he repeated the description, finishing "And that's how we solve it." "The Wall" as it became known, became a focal image for the group.

5.1. How they occurred

In the observed examples, the mental imagery used by a key team member in constructing an abstract solution to a design problem was externalized and adopted by the rest of the team as a focal image. The images were used both to convey the proposed solution and to co-ordinate subsequent design discussions. The examples all occurred in the context of design, and the images concerned all or a substantial part of the proposed abstract solution.

5.2. The nature of the images

The images tend to be some form of analogy or metaphor, depicting key structural abstractions. But they can also be 'perspective' images: 'if we look at it like this, from this angle, it fits together like this'—a visualization of priorities, of key information flows or of key entities in relationship. The image is a *conceptual* configuration that may or may not have any direct correlation to eventual system configuration.

5.3. The process of assimilation

In all of the examples observed, the image was initially described to other members of the team by the originator. Members of the team discussed the image, with rounds of 'is it like this' in order to establish and check their understanding. Although initial questions about the image were inevitably answered by the originator, the locus did shift, with later questions being answered by various members of the team as they assimilated the image. The image was 'interrogated', for example establishing its boundaries with questions about 'how is it different from this'; considering consequences with questions like 'if it's like this, does it mean it also does that?'; assessing its adequacy with questions about how it solved key problems; and seeking its power with questions about what insights it could offer about particular issues. In the course of the discussion and interrogation, the image might be embellished—or abandoned.

5.4. They are sketched

Sketching is a typical part of the process of assimilation, embodying the transition from 'mental image' to 'external representation'. The sketches may be various, with more than one sketch per image, but a characteristic of a successful focal image is that the 'mature' sketches of it are useful and meaningful to all members of the group. This fits well with the literature about the importance of good external representations in design reasoning (e.g., [47,48], and others). Ko et al. [49], in a study of the information needs of software developers, found that design questions about intent and rationale were among the most difficult to satisfy. These 'mature' sketches, with their shared interpretation, provide a means of preserving intent and rationale within the team.

5.5. Continuing role reflected in team language

If the image is adopted by the team, it becomes a focal point of design discussions, and key terms or phrases relating to it become common. Short-hand references to the image are incorporated into the team's jargon to stand for the whole concept. But the image is 'team-private'; it typically does not get passed outside the team and typically does not reach the documentation. Although the 'mature' sketches noted above might be strong candidates for documentation, as they embody tested collective understanding, whether that understanding would persist over time and outside the co-ordination process is an open question. Ko et al. [49] highlight that "Even when developers found a person to ask, identifying the information they sought was hard to express…" The short-hand references may limit the utility for persistent documentation, creating issues of interpretation for down-stream developers who were not part of the coordinated design and development.

5.6. Imagery as a co-ordination mechanism

The images discussed and interrogated by the team provide a co-ordination mechanism. Effective co-ordination will by definition require the use of images that are meaningful to the members of the group. The literature on schema provides explanation here (e.g., [20]). Co-ordination – meaningful discourse – requires shared referents. If there is a shared, socially agreed schema or entity, this can be named and called into play. But what happens when the discourse concerns an invention, an innovation, something for which there is no existing terminology, no pre-existing schema? A preverbal image in the mind of the originator, if it cannot be articulated or named, is not available to the group for inspection and discussion. The use of extended metaphor, with properties in several different facets, provides a way of establishing a new schema. In describing the image, the originator is establishing common reference points. The recipient chooses what is salient in the properties of interest. By interrogating and discussing the image and its implications with the originator, the recipients are establishing a shared semantics, and the originator is co-ordinating with the rest of the team (cf. Shadbolt's research [50] on people's use of maps and the establishment of a common semantics). The discussion of the metaphor allows the team to establish whether they understand the same thing as each other. The establishment of a richly visualized, shared image (and the adoption of economical short-hand references) facilitates keeping the solution in working memory (e.g. [34]).

It is interesting to note that this co-ordination issue has been taken on board by recent software development methodologies, which often try to address it by creating an immersive environment of discourse and artefacts which is intended to promote regular re-calibration with the other team members and with artefacts of the project. For example, 'contextual design' [51] describes 'living inside' displays of the external representations in order to

internalize the model (to take it into one's thinking), referring to the displayed artefacts as "public memory and conscience". In another example, 'extreme programming' [52] emphasizes the importance of metaphor, requiring the whole team to subscribe to a metaphor in order to know that they are all working on the same thing. In that case, the metaphor is carried into the code, for example through naming.

So, individual imagery does sometimes enter external interaction. The mental imagery used by a key team member in constructing an abstract solution to a design problem can in some cases be externalized and adopted by the rest of the team as a focal image. Discussing, sketching and 'interrogating' the image helps the team to co-ordinate their design models so that they are all working on the same problem—which is fundamental to the effective operation of the team.

6. Study 3: To what extent do these software developers use available visualization tools?

Given the naturally occurring use of image, abstraction, and sketches and other external design representations, to what extent do these software developers use available visualization tools to help them? This section reports on opportunistic interviews with proficient software developers about their use of (or reluctance to use) available software visualization tools.

Study design—interviews of 12 software developers in 3 companies, and review of practice: Semi-structured interviews were conducted with 12 software developers in high-performing teams, from 3 different companies. This study was conducted in the shadows of other studies into other aspects of software development (as described in Section 5 above). Key team members were interviewed—those likely to make decisions on models and solutions as well as decisions on tools. The interviews included a 'guided tour' of their libraries of applications that included software visualization. The 'tours' were led by the informants. Questions were asked only when the protocol was not covered spontaneously. For each relevant application, information was collected regarding the following:

A general description, including what sorts of visualizations it afforded;

How it had been used, for which tasks, with examples if available;

To what extent it had been used;

If it was still in use, what it was considered to be 'good for';

If it had been abandoned, why it had been set aside.

Detailed notes were taken (including verbatim quotation), and audio recordings were made where permitted. The face-to-face semi-structured interviews were augmented with telephone and email queries to additional informants. Again, the analysis was inductive, identifying patterns and common elements among the accounts. Material from additional informants was not used in the initial analysis, but was used as a validation sample for the observations drawn from the principal data.

The software developers talk about software visualization with respect to three major activities: comprehension (particularly comprehension of inherited code), debugging, and design reasoning. These software developers showed *no* reluctance to investigate potential tools, and they described trials of tools as diverse as Burr-Brown DSP development package, Cantata, MatLab, Metrowerks Code Warrier IDE, Mind Manager, MS Project, MS-Select, Rational Rose, Software through Pictures (STP), and Visio (among others). Tools they did take up usually related to software comprehension and debugging. In that context, the tools that persisted were those that were robust, worked at scale, and associated well with their preferred software development environment. But even in debugging and comprehension, the experts relied more on their own systematic practices than on visualizations—and their use of available visualizations related to how directly the visualizations supported their practices. Tools which simply re-presented available information, which failed to provide forms of abstraction or selection, or which embodied assumptions at odds with the experts' practices were discarded.

Take-up was extremely low in the context of design and generation. The exception was for general tools such as MatLab, which allowed software developers to realize their own visualizations (as is discussed in the next section)—in effect for visualization-builders rather than visualizations *per se*. So what makes other tools into shelfware? (Please note that 'Not invented here' was never offered as a reason not to use a tool.)

reliability: Packages that crashed or misbehaved on first exposure did not usually get a second chance.

overheads: The cost of take-up was perceived as too high. Usually the cost was associated with taking on the philosophies, models or methodologies embodied in and enveloping the visualization elements. Where there were major discrepancies of process between the package and existing practice, or where there was incompatibility between the package and other tools currently in use, the package was typically discarded as likely to fail. It must be considered that these are high-performance teams, with well-established methodologies and work practices. They continually seek tools and methods that augment or extend their practice, but they are reluctant to change work practices (particularly work style) without necessity.

lack of insight: Tools that simply re-present available information (e.g., simplistic diagram generation from program text) do not provide any insight. Experts seek facilities that contribute to insight, e.g., useful abstractions, ready juxtapositions, information about otherwise obscure transformations, informed selection of key information, etc.

lack of selectivity: Many packages produce output that is too big, too complicated, or undifferentiated. For example, all processes or all data are handled in the same way, and everything is included. Experts want ways of reasoning about artefacts that are 'too big to fit in one head'; simply repackaging massive textual information into a massive graphical representation is not helpful. They seek tools that can provide a useful focus on things

that are 'too big', that can make appropriate and meaningful selections for visualization.

lack of domain knowledge: Most tools are generic and hence are too low-level. Tools that work from the code, or from the code and some data it operates on, are unlikely to provide selection, abstraction, or insight useful at a design level, because the information most crucial to the software developer – what the program represents, rather than the computer representation of it – is not in the code. At best, the software developer's intentions might be captured in the comments. As the level of abstraction rises, the tools needed are more specific, they must contain more knowledge of the application domain. Experts want to see software visualized in context—not just what the code does, but what it means.

assimilation: Sometimes tools that do provide useful insights are set aside after a period of use, because the need for the tool becomes 'extinct' when the function (or model or method) the tool embodies is internalized by the software developer.

What the software developers seek in visualization tools (selectivity, meaningful chunking or abstractions) – especially in tools for design reasoning – appears to relate directly to their own mental imagery, which emphasizes selectivity, focus, chunking and abstraction. It relates as well to what the literature has to tell us about ways in which human beings deal with massive amounts of information, e.g. chunking, selection, schema—and to the nature of the differences between experts and novices. They want (and as the next section will indicate, they build) tools that have domain knowledge and can organize visualizations according to *conceptual structure*, rather than physical or programme structure—but that can maintain access to the mapping between the two. For example, the software developers talk about tracking variables, but at a *conceptual* rather than a code level. A 'conceptual variable' might encompass many elements in the software, perhaps a number of data streams or a number of buffers composing one thing, with the potential for megabytes of data being referred to as one object. The software developers distinguish between 'debugging the software' (i.e., debugging what is written) and 'debugging the application' (i.e., debugging the design, what is intended).

7. Study 4: The visualizations experts build for themselves

So, what sorts of visualization tools do experts build for themselves, and what relationship do they have to experts' mental imagery?

Study design—opportunistic observation over many years: multiple companies, multiple projects: Over a number of years, in association with other studies of software development, we asked experts and high-performing teams to show us the visualization tools they built for themselves. In some cases, the demonstrations were volunteered (for example, in the mental imagery study, when one expert in describing a form of imagery said, 'here I can show you'; or for example in the survey of application use, when some software developers included their own tools in the review of tools they'd tried). Examples were collected to the extent permitted by the informants; access was sensitive because of the proprietary nature of the material examined, and the examples offered below are used with permission. Each example was documented in field notes, accompanied when possible by screen dumps annotated by the developers. The visualization developers were asked to demonstrate the visualization, and they were interviewed about their intentions for and use of the visualization. Again, the analysis was inductive, seeking patterns and lessons across the examples. And, again, the findings were presented to and discussed with the informants, who were asked to scrutinize the accounts and their analysis critically, identifying any inaccuracies.

The experts' own visualizations tended to be designed for a specific context, rather than generic. In one expert's characterization of what distinguished his team's own tool from other packages they had tried: ''the home-built tool is closer to the domain and contains domain knowledge''. The tools appeared to fall into two categories, corresponding to the distinction the experts made between 'debugging the software' and 'debugging the application'. Each category is discussed in turn.

7.1. Low-level aspect visualization

A typical visualization tool in this class is one team's 'schematic browser' (Fig. 1). This program highlighted objects and signal flows with colour. It allowed the user to trace signals across the whole design (i.e., across multiple pages), to move up and down the hierarchy of objects, and to find and relate connections. It moved through the levels of abstraction, relating connections at higher levels to connections lower down through any number of levels and through any number of name changes, for example following one conceptual variable from the top to the bottom of a dozen-deep hierarchy and automatically tracking the name changes at different levels. It allowed the user to examine the value of something on a connection, and to manipulate values, for example altering a value on one connection while monitoring others and hence identifying the connective relationship between different parts. The program embodied domain information, for example having cognizance of the use of given structures, in effect having cognizance of what they represented, of the conceptual objects into which schematic elements were composed.

It appears that these tools reflect some aspects of what the imagery presents, but they do not 'look like' what the engineers 'see' in their minds. There are a number of such tools, especially ones that highlight aspects of circuits or code (e.g., signal flows, variables) or tools for data visualization, as well as tools that represent aspects of complexity or usage patterns. In effect, they visualize things engineers need to take into account in their reasoning, or things they need in order to form correct mental models, rather than depicting particular mental images.

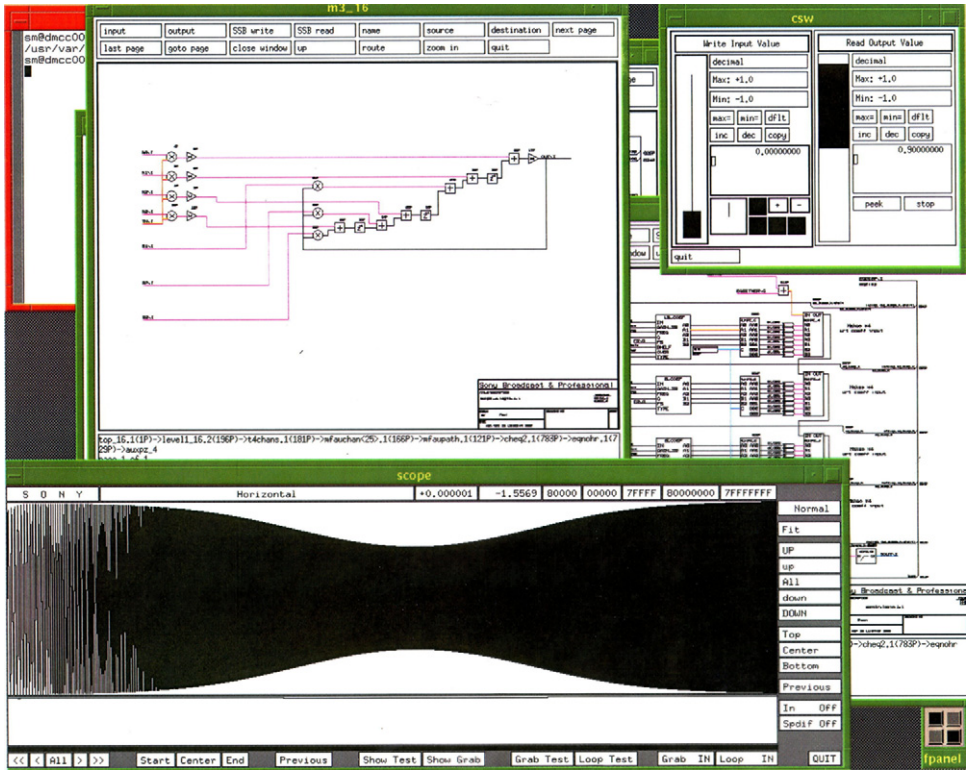

Fig. 1. A 'schematic browser'—an example of low-level aspect visualization.

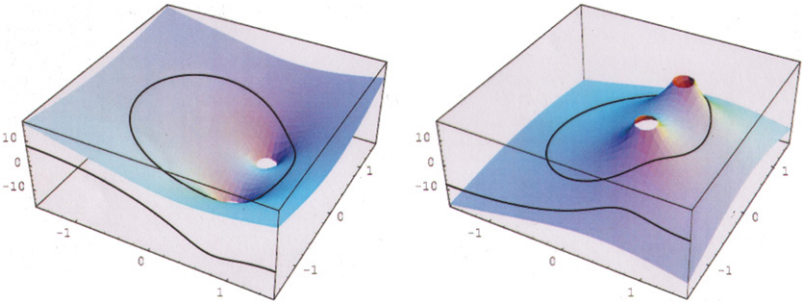

Fig. 2. The 'rubber sheet'—an example of concept visualization.

7.2. Conceptual visualization

A typical visualization tool in this class is one team's 'rubber sheet' (Fig. 2). This program is a visualization of a function used to build digital electronic filters for signals, which normally have a very complex and non-intuitive relationship between the values in the equations and the effect they have on the frequency response of the resulting filter. The tool allows the user to design the frequency response visually by determining the profile of a single line across the 'rubber sheet'. The user moves peaks and dips in a conforming surface, rather than changing otherwise random-looking values in equations. The altitude one encounters on a walk through the resulting terrain along a particular path determines the final frequency response. The insight comes from the 'terrain' context. If one is just moving the points where the peaks and dips are, and can only see the values along the line, it is hard to see how the values along the line are varied by moving the peaks and dips. However, if one can see the whole terrain, it is easy to comprehend why the peaks and dips have moved the fixed path up and down. Designers do not need to know all this terrain information in order to know what the filter does, but it provides the link between what the filter does and what the designer can control.

It appears that these tools can be close to what engineers 'see' in their minds. (Indeed, this example was demonstrated as a depiction of one software developer's personal mental imagery.) As in the example, they often bear strong resemblance to mathematical visualizations or illustrations.

The two categories of tool differ not just in their relationship to experts' mental imagery, but also in how they are used. The low-level aspect visualizations tend to be used to debug the artefact. They pre-suppose that the expert's understanding of the artefact is correct, and examine the artefact in order to investigate its behaviour. The conceptual visualizations tend to be used to debug the concept or process—to reason about the design.

8. Implications and discussion

It has been suggested that the big issues that face software visualization – particularly with respect to design – relate to matching visualizations to human needs. What can we and can we not visualize? Are we visualizing the right things? Currently, it is still arguable that what is visualized is what can be visualized, not necessarily what needs to be visualized. The big technical challenges lie in developing the analysis and selection techniques needed to tailor visualizations to support human cognition. Tools that simply re-present available information (e.g., simplistic diagram generation from program text) do not provide insight. Software developers seek facilities that contribute to insight, e.g., useful abstractions, ready juxtapositions, information about otherwise obscure transformations, informed selection of key information, etc. Visualization developers are still struggling with what Gómez-Henríquez [53] calls the 'probe effect': ensuring that the visualization is reliable enough to ensure that what the user sees is what is really happening—whether or not it is what the user needs to see. Underlying the challenge to identify the most appropriate visualization for a given task is the need to reach a well-founded understanding of what makes visualizations appropriate for given tasks.

The features observed in expert practitioner behaviour in this domain are consistent with findings in a range of related literatures. What are the implications, and what practical advice can be offered as a result of these literatures?

8.1. Visualizing concepts and intentions vs. re-presenting implementations

There are few visualizations yet to support conceptual design. There is a need to provide conceptual visualizations, rather than just re-presentation of the code, performance or data flow. This highlights the need to make available information that is not typically contained in the source code: information about the originators' intentions and models of the software. This implies that the visualizations (and the tools that drive them) must embody more knowledge of the application domain. As discussed in Section 6, experts want to see software visualized in context—not just what the code does, but what it means.

8.2. Intelligent selection requires domain knowledge

The utility of visualization lies not in mere re-presentation of data, but in an appropriate and meaningful distillation and abstraction of the data in order to provide access to desired information about the software. That is, it is no good translating massive source code into an equally massive visualization; what is required is views on the artefact that disclose significant patterns within it. Tudoreanu [54] discusses software visualization in terms of "cognitive economy": minimizing cognitive load by reducing the amount of information handled by the user and maximizing the information pertinent to the user problem (which is different from reducing complexity related to visual displays). Cognitive economy requires that the visualization be customized for the problem at hand, tailored to the user's task and goals. This, in turn, requires some knowledge of the domain.

According to Chi, "…good visualizations are coupled with good analysis algorithms. We can get the most power out of visualization if we use a sophisticated analysis computation that distils the data further from the raw data."[55]

One implication is that generic tools are not selective. Because they do not contain domain knowledge, they cannot depict what the software developers actually reason about when they reason about design. Automatic generation from code is inherently unlikely to produce conceptual visualizations because the code does not contain information about intentions and principles. The extent to which domain knowledge can be encoded is a suitable topic for further research.

8.3. Design visualization versus program visualization

The distinction between low-level aspect visualization and conceptual level visualization (in the self-built tools) is also important. At the feature level the visualization contributes to the mental imagery rather than reflecting it. At the conceptual level, in contrast, it appears that there can be a more direct relationship between the mental imagery and the software visualization. More work is also needed on design visualizations (as opposed to software visualizations) and on the interaction between the two – to what extent does understanding design visualization contribute to solving problems in the domain of program visualization?

It is important to remember that there are differences between design visualization and program visualization. Design visualization is a divergent thinking problem in the early stages at least, which requires creativity and readiness to think 'outside the box'. Schön [48] talks about a design as a 'holding environment' for a set of ideas. The importance of fluidity, selectivity, and abstract structure is emphasized both by the experts' own mental imagery and by their stated requirements for visualization tools. It is little surprise that, in this context, experts conclude that "NOBO [whiteboard]...The best of all tools until the pens dry out. No question." and "Nothing is as good – and as quick – as pencil and paper." Program visualization, in

contrast, often involves dealing with existing legacy systems, where an important part of the task is reconstructing the design reasoning of previous software developers, which led to the system under investigation—this paper contributes little to legacy system comprehension.

9. Conclusion

It appears that, in the context of the design and generation of 'engineering software', there is sometimes a fairly direct relationship between mental imagery and software visualization—but that it is more often the case that visualizations contribute to rather than reflect mental imagery. It also appears that the externalization of expert mental imagery can play an important role in the design reasoning of high-performance teams, both through coordination of team design discussions and through embodiment in custom visualization tools. Experts tend not to use available visualization tools, because they do not contribute sufficiently to design reasoning. Their custom visualization tools differ from others in their embodiment of domain knowledge, facilitating investigations at a conceptual level.

Acknowledgements

This paper was first presented as a keynote address at the Second Program Visualization Workshop, Hornstrup Centret, Denmark, in June 2002. It is produced here with kind permission of the workshop chair, Prof. Mordechi Ben-Ari. The author is profoundly grateful to the expert software developers, without whom the paper would not have been possible, and to their companies which permitted access. Thanks are due to colleagues who provided critical commentary, including Alan Blackwell, Peter Eastty, Marc Eisenstadt, Henrik Gedenryd, Simon Holland, William Kentish, Clayton Lewis, Hugh Robinson, Jennifer Rode, and Helen Sharp—as well as the anonymous referees. Special thanks are due to Gordon Rugg, who was instrumental in writing the paper. Some of the observations were conducted under EPSRC grant GR/J48689 (Facilitating Communication across Domains of Engineering). Others were conducted under an EPSRC Advanced Research Fellowship AF/98/0597. The author is a Royal Society Wolfson Research Merit Award Holder.

References

[1] R.W. Hamming, Numerical Methods for Scientists and Engineers, 2nd ed., Dover Publications, 1987.
[2] M. Petre, A. Blackwell, T.R.G. Green, Cognitive questions in software visualization, in: J. Stasko J. Domingue, M. Brown, B. Price (Eds.), Software Visualization: Programming as a Multimedia Experience, MIT Press, 1998, pp. 453–480.
[3] V. Goel, Sketches of Thought, MIT Press, 1995.
[4] J.I. Maletic, A. Marcus, M.L. Collard, A task oriented view of software visualization, in IEEE First International Workshop on Visualizing Software for Understanding and Analysis, IEEE Computer Society Press, 2002, pp. 32–40.
[5] R. Baecker, A. Marcus, Human Factors and Typography for More Readable Programs, ACM Press, 1990.
[6] P. Eades, K. Zhang (Eds.), Software Visualization, World Scientific Publications, 1996.
[7] J. Stasko, J. Domingue, M.H. Brown, B.A. Price (Eds.), Software Visualization: Programming as a Multimedia Experience, MIT Press, 1998.
[8] S. Diehl (Ed.), Software Visualization, Springer, 2002.
[9] K. Zhang (Ed.), Software Visualization: From Theory to Practice, Kluwer Academic Publishers, 2003.
[10] B.A. Price, R. Baecker, I.S. Small, A principled taxonomy of software visualization, Journal of Visual Languages and Computing 4 (3) (1998) 211–266.
[11] SCG Smallwiki CodeCrawler: CodeCrawler: a non-exhaustive list of software visualization tools' 〈http://smallwiki.unibe.ch/codecrawler/anon-exhaustivelistofsoftwarevisualizationtools/〉. [Accessed June 2006].
[12] S. Lammers, Programmers at Work, Microsoft Press, 1986.
[13] S. Kaplan, L. Gruppen, L.M. Leventhal, F. Board, The Components of Expertise: A Cross-Disciplinary Review, The University of Michigan, 1986.
[14] C.M. Allwood, Novices on the computer: a review of the literature, International Journal of Man-Machine Studies 25 (1986) 633–658.
[15] M.T.H. Chi, P.J. Feltovich, R. Glaser, Categorization and representation of physics problems by experts and novices, Cognitive Science 5 (1981) 121–152.
[16] M. Weiser, J. Shertz, Programming problem representation in novice and expert programmers, International Journal of Man-Machine Studies 19 (1983) 391–398.
[17] J.H. Larkin, The role of problem representation in physics, in: D. Gentner, A.L. Stevens (Eds.), Mental Models, Lawrence Erlbaum, 1983.
[18] R. Jeffries, A.A. Turner, P.G. Polson, M.E. Atwood, The processes involved in designing software, in: J.R. Anderson (Ed.), Cognitive Skills and Their Acquisition, Lawrence Erlbaum, 1981, pp. 255–283.
[19] B. Adelson, E. Soloway, The role of domain experience in software design, IEEE Transactions on Software Engineering SE-11 (11) (1985) 1351–1360.
[20] F.C. Bartlett, Remembering: An Experimental and Social Study, Cambridge University Press, 1932.
[21] M.T.H. Chi, R. Glaser, M.J. Farr (Eds.), The Nature of Expertise, Lawrence Erlbaum, 1988.
[22] H.A. Simon, The structure of ill-structured problems, Artificial Intelligence 4 (1973) 181–202.
[23] C. Cole, C.C. Kuhlthau, Information and information seeking of novice versus expert lawyers: how experts add value, The New Review of Information Behaviour Research 2000 (2000) 103–115.
[24] R. Tourangeau, R. Sternberg, Understanding and appreciating metaphors, Cognition 11 (1982) 203–244.
[25] M. Petre, Why looking isn't always seeing: readership skills and graphical programming, Communications of the ACM 38 (6) (1995) 33–44.
[26] H. Ebbinghaus, Memory: A Contribution to Experimental Psychology (Henry A. Ruger, Clara E. Bussenius, trans.), New York Teachers College, Columbia University, 1913.
[27] A.R. Luria, The Mind of a Mnemonist: A Little Book about a Vast Memory (Lynn Solotaroff, trans.), Basic Books, 1968.
[28] E.F. Loftus, J.C. Palmer, Reconstruction of automobile destruction: an example of the interaction between language and memory, Journal of Verbal Learning and Verbal Behaviour 13 (1974) 585–589.
[29] D. Kieras, Beyond pictures and words: alternative information-processing models for imagery effects in verbal memory, Psychological Bulletin 85 (3) (1978) 532–554.
[30] K. Mani, P.N. Johnson-Laird, The mental representations of spatial descriptions, Memory and Cognition 10 (2) (1982) 181–187.
[31] S.J. Payne, Memory for mental models of spatial descriptions: an episodic-construction-trace hypothesis, Memory and Cognition 21 (5) (1993) 591–603.
[32] R.E. Anderson, T. Helstrup, Visual discovery in mind and on paper, Memory and Cognition 21 (3) (1993) 283–293.
[33] F.C. Bartlett, The relevance of visual imagery to thinking, British Journal of Psychology 18 (1) (1927) 23–29.
[34] R.H. Logie, Characteristics of visual short-term memory, European Journal of Cognitive Psychology 1 (1989) 275–284.
[35] R.K. Lindsay, Images and inference, Cognition 29 (3) (1988) 229–250.
[36] B. Adelson, D. Littman, K. Ehrlich, J. Black, E. Soloway, Novice-expert differences in software design, in: Interact '84: First IFIP Conference on Human–Computer Interaction, Elsevier, 1984.

[37] S.K. Card, J.D. Mackinlay, B. Shneiderman, Information visualization, in: S.K. Card, J.D. Mackinlay, B. Shneiderman (Eds.), Readings in Information Visualization: Using Vision to Think, Morgan Kaufmann Publishers Inc., 1999, pp. 1–34.
[38] L.J. Ball, T.C. Ormerod, Putting ethnography to work: the case for a cognitive ethnography of design, International Journal of Human–Computer Studies 53 (2000) 147–168.
[39] B. Curtis, H. Krasner, N. Iscoe, A field study of the design process for large systems, Communications of the ACM 31 (11) (1988) 1268–1287.
[40] J.W. Schooler, S. Ohlsson, K. Brooks, Thoughts beyond words: when language overshadows insight, Journal of Experimental Psychology: General 122 (2) (1993) 166–183.
[41] K.A. Ericsson, H.A. Simon, Protocol analysis: Verbal reports as data, MIT Press, 1985.
[42] A.N. Katz, What does it mean to be a high imager?, in: J.C. Yuille (Ed.), Imagery, Memory and Cognition: Essays in Honor of Allan Paivio, Erlbaum, 1983.
[43] C.H. Ernest, Imagery ability and cognition: a critical review, Journal of Mental Imagery 1 (2) (1977) 181–216.
[44] M. Petre, A. Blackwell, A glimpse of programmers' mental imagery, in: S. Wiedenbeck, J. Scholtz (Eds.), Empirical Studies of Programmers: Seventh Workshop, ACM Press, 1997, pp. 109–123.
[45] B.G. Glaser, A.L. Strauss, The Discovery of Grounded Theory: Strategies for Qualitative Research, Aldine, 1967.
[46] M. Petre, Team coordination through externalised mental imagery, International Journal of Human–Computer Studies 61 (2) (2004) 205–218.
[47] N.V. Flor, E.L. Hutchins, Analysing distributed cognition in software teams: a case study of team programming during perfective software maintenance, in: J. Koenemann-Belliveau, T.G. Moher, S.P. Roberston (Eds.), Empirical Studies of Programmers: Fourth Workshop, Ablex, 1991.
[48] D. Schön, Design rules, types and worlds, Design Studies 9 (3) (1988) 181–190.
[49] A.J. Ko, R. DeLine, G. Venolia, Information needs in collocated software development teams, in: 29th International Conference on Software Engineering (ICSE '07), IEEE Computer Society, 2007, pp. 344–353.
[50] N.R. Shadbolt, Constituting reference in natural language: the problem of referential opacity, Ph.D. Thesis, University of Edinburgh, 1984.
[51] H. Beyer, K. Holtzblatt, Contextual Design: Defining Customer-Centered Systems, Morgan Kaufmann, 1988.
[52] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.
[53] L.M. Gómez Henríquez, . Software visualization: an overview, Informatik 2 (2001) 4–7.
[54] M.E. Tudoreanu, Designing effective program visualization tools for reducing user's cognitive effort, ACM Symposium on Software Visualization, vol. 213, ACM Press, 2003, pp. 105–114.
[55] E.H. Chi, The Future of Software: Visualization+Computation Tools, Future of Software Special Issue, Fawcette Technical Publishing, 2000.