uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Adam Murray

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Ph.D. (Computer Science)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Discourse Structure of Software Explanation: Snapshot Theory, Cognitive Patterns and Grounded Theory Methods

TITRE DE LA THÈSE / TITLE OF THESIS

Timothy Lethbridge

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Daniel Amyot

Liam Peyton

Dwight Deugo

Pierre-N. Robillard

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

# Discourse Structure of Software Explanation: Snapshot Theory, Cognitive Patterns and Grounded Theory Methods

by

Adam Ross Murray
B.Sc., University of Ottawa, 2000

Doctoral Thesis

Presented to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree

**Doctor of Philosophy (Computer Science*)**

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, K1N 6N5
Canada



* The Ph.D. program in Computer Science is a joint program with Carleton University, administered by the Ottawa Carleton Institute for Computer Science

*Your file Votre référence*
*ISBN: 978-0-494-25888-0*
*Our file Notre référence*
*ISBN: 978-0-494-25888-0*

## Abstract

This thesis introduces a grounded theory of the discourse structure that professional software engineers use while explaining software. The 'Snapshot Theory' relates how the *snapshot* is the critical moment and fundamental building block in the discourse structure. We built Snapshot Theory by applying a rigorous qualitative data analysis research methodology, known as grounded theory, on observational data of professional software developers explaining software architectures. We developed a research methodology, qualitative analysis tools and case data in support of our investigation. We present two versions of our theory, the grounded theory tied to evidence, and our interpretation of the applied theory in pattern form, as *cognitive patterns*. We intend cognitive patterns to facilitate the development of software tool features based on Snapshot Theory.

## Acknowledgements

When I began this work, I could not have anticipated that my work would draw to a close so many times only to begin anew. In the autumn of 2001, as I contemplated how to complete my Masters thesis, my mentor and friend, Professor Timothy Lethbridge invited me to extend my work (which had just begun) into a doctoral thesis. In the (ever) spring of Victoria in 2003, Peggy Storey provided invaluable guidance and support and Hausi Muller taught me about *fun* research; I discovered my relentless passion for the land and I reinvented my 'nearly complete' research – a fresh start again. Spring of 2005, thanks to the advice and support of Janice Singer my work had *truly* only just begun. And with the thesis confidently written and successfully defended before my honourable committee, to whom I am deeply indebted, I have finally and beyond doubt reached the beginning of my work.

I am proud of this thesis, and thus I am deeply grateful to the following people who helped me to extend my mind:

- I am thankful for the funding for my research from NSERC, OGS, the Consortium for Software Engineering Research (CSER), IBM, Mitel Corporation, the University of Ottawa, National Research Council Canada, and Timothy Lethbridge.
- CSER researchers and CSER alumni are an incredible crew: *excelsior*.
- Thanks to Andrew Walenstein for our dialogue and solicitous review of my work.
- My thanks to the IBM Centre for Advanced Studies, and special thanks to Marcellus Mindel. I also thank Kevin Grignon and Susan McIntyre.
- My appreciation to Lisa Legault and Nash Majstronovic who reassured me that snapshots are of real psychological interest. Likewise, Nick the cook and Eric Budai showed me magic that made a huge difference in my project success. My thanks also to Faune for her overall support and help with transcription.
- My sincere thanks to the unnamed participants in my study from IBM and Mitel.
- I am obliged to the attendees of my Pattern Writing Workshop, with regards to Richard Gabriel and Bob Hanmer. I am especially appreciative for the sage shepherding of Linda Rising.
- I am so grateful for the unbounded love and support of my mother, Joan Murray.

# Contents

## List of Figures



## List of Tables

# Chapter 1 Introduction

In this thesis, our primary contribution to the software engineering literature is a theory[2] that describes how software engineers explain software. We call this *Snapshot Theory.* A snapshot represents an interval or moment in the development of a software model when the model is cohesive or conceptually whole. This thesis builds on prior research [13, 61, 80, 112] in which researchers derived the comprehension processes of software developers from the developers' explanations.

The research presented in this thesis will be useful to software tool developers because designing *is* explaining: a deeper appreciation of how developers explain software through snapshots can lead to improvements in software tool design and software processes. The research also contributes the field of program understanding. Other important contributions include the qualitative methodology we used to develop Snapshot Theory, a qualitative analysis tool we call Qanal, and the presentation of Snapshot Theory as *cognitive patterns.*

We will outline each of these contributions in the following subsections, but we begin with a short fictional narrative to illustrate the importance of Snapshot Theory:

> *While attending a data structures course, a student named Ann cannot understand her professor's software explanation. As her professor* models *the procedure for the insertion of a node at the end of a list, Ann reconstructs his model in her notes. In preparation for her final examination, Ann reviews her notes but cannot fathom the meaning of the model. Ann asks her peer Andrew for help and Andrew provides Ann with his notes for her study. Ann for the first time understands the previously foreign concepts. She is taken aback by Andrew's note-taking approach: by tentatively observing the discussion before a critical moment, Andrew recorded* snapshots *in the professor's explanation. As Ann compares her notes with Andrew's, she realises the professor's final representation lacks* details *she needs to effectively understand it; snapshots provide these details. Andrew explains how he could identify snapshots based on the behaviour of the professor. Ann improves her approach to learning.*

[2] See glossary, pg. 180, for a definition of "theory".

*When Ann graduates, she joins a telecommunications company. The regularity of software explanation in* informal whiteboard sessions *astounds her. While in school, she did not anticipate this level of interaction among peers. Informal whiteboard sessions highlight the software architecture (i.e. high-level aspects of the systems such as the overall organization, the decomposition into components, the assignment of functionality to components, and the way the components interact [18]). She cannot work two days without engaging a peer in a design discussion where the whiteboard is absolutely necessary. What is particularly striking to Ann is that snapshot behaviour is exhibited by* all *of her peers. Even when her peers provide* lateral *or* weak *snapshots, Ann recognizes the existence of the snapshots and thus enriches her understanding of a complex telecommunication system.*

*When Ann leaves the telecommunications company for another company that develops software tools for professional software developers, she is not surprised to discover snapshot behaviour in another organization. She also discovers that snapshots relate to one another and relate to a more complete snapshot that contributes to high-level understanding. Ann realises that snapshot behaviour is pervasive and wonders if software tool features could better support snapshot behaviour. Since this challenge is open-ended, a concrete methodology would help her execute her investigation.*

This dissertation addresses a similar challenge.

## 1.1 The First Problem: Software Explanation

*What is the discourse structure by which software professionals explain software*?

This question is central to this thesis and answering it may facilitate the design of software tools that explicitly support that structure. If we can uncover a discourse structure that developers naturally use, we can build tools grounded in developers' natural behaviour.

The main contribution of this thesis is *Snapshot Theory*, in which the "snapshot" is a critical moment in the discourse structure. In the development of this theory, we asked the following research questions:

- What cues can allow an observer to identify a snapshot?
- What is the composition of the discourse structure leading up to the snapshot?
- How does this discourse structure leading up to a snapshot begin?
- How does an observer know when the snapshot has occurred?
- How does the snapshot contribute to other snapshots?
- How does the snapshot contribute to the overall explanation?

## 1.2 The Second Problem: The Methodology to be Used

The formation of the above research questions led us to a second problem. *What methodology will we use to enable us to answer these research questions*?

To address this problem, we used qualitative research methods that incorporate concepts of the *grounded theory* method typically used in sociological research. Grounded theory is a rigorous process for the development of theories from observational data, particularly observations of people. We propose that applying a deep, evidence-based understanding of human activities in software engineering can lead to the creation of better tools and processes.

We designed a process that involves the use of multiple stages of data collection, as well as the refinement and interrelationship of categories of information. The data-capture phase of our study consisted of videotaped interview sessions on-site at Mitel Corporation and IBM. The goal of data collection is to gather open-ended, emerging data. Our process includes several analytical steps that traverse the process from theory building to theory presentation. The results (or "outputs") of each research step are inputs for the next step. The steps are as follows: data preparation, initial analysis, topical analysis, category analysis and theory building.

In Chapter 3, we present the pilot study we used to uncover many of the challenges we faced in developing our methodology. Our complete methodology is presented in Chapter 4 and constitutes a distinct contribution to the empirical software engineering community.

It is worthwhile to note that when we started our inquiry, we did not have the above problems in mind. We were initially interested in studying the diagrams and diagrammatic techniques used by professional software developers; in particular we were interested in the

extent to which they tend to use the Unified Modeling Language (UML) or a hybrid of diagrammatic techniques not found in UML. Only once we started to gather data did we realise its richness, and the need to understand its discourse structure. The latter became our primary problem.

To solve the primary problem we needed to develop a rigorous methodology (our second problem), software instrumentation to support our qualitative analysis, which we call Qanal, and a succinct way to present our results (our third problem, discussed below). Our investigation led us to generate an abstract theory of discourse; in other words, our interest broadened to include elements of communication other than just diagrams.

## 1.3 The Third Problem: Reporting the Results

While building our Snapshot Theory, we encountered a third problem: *how will we report our results in order to be both scientifically valid and accessible to readers*? There are two sides to the exposition of theoretical work in software engineering. First, the researcher's perspective: software researchers must know the genesis of a theory so they can assess its validity[3]. Second, the practitioner's perspective: software professionals require generalized yet practical approaches that allow them to apply the theories in a meaningful way.

Part of the results-reporting problem derives from the approach widely used by those of us researching software engineering tools. We build tools, and sometimes these are based on empirical studies that yield theories of user cognition, but the theories are embedded in the tools themselves. Sometimes we publish papers based on either the theories or the tools, but these may focus on describing the research for the consumption of other academics, rather than making it accessible to the tool designer. Aside from some broad guidelines, there is a weak link between the ideas and the actions or considerations the designer requires to build tools, or tool features, that are based on these ideas. As a result, the designer builds tool features that exhibit theory replicas. The replicas are superficially similar to the original theories, but do not capture the essence or deep understanding of the original. This recalls cargo-cult design [50]; Melanesians used straw

[3] In contrast with sociological research, where there is more reverence for the text of the theory –theories in that discipline are based more on their face value

and bamboo to build imitation airports, control towers and landing strips, reasoning that if they built exact replicas of the white man's artefacts, they would receive the same benefits – the cargo.

To address these problems in a research environment requires the careful balance of two different groups of practitioners. Firstly, academics require methodological rigour and transparent results. How can scientists otherwise demonstrate that cognitive theories are well-grounded? Secondly, professionals require simple but appropriate guides to apply the cognitive theories in an appropriate context. Clearly, these dual requirements can be at odds. In Chapters 5 and 6, we reconcile these ideas. In Chapter 5, we present our theory as mandated by the grounded theory approach, and in Chapter 6 we present the application as *cognitive patterns*.

Cognitive patterns are patterns[4] inspired by the work of Christopher Alexander [8, 9] and analogous to the notion of design patterns [52] – patterns of which the software engineering community is already aware. We define a cognitive pattern as follows: A cognitive pattern is a structured exposition of a solution to a recurring cognitive problem in a specific context through which readers gain understanding.

## 1.4 Summary

In this thesis we tackle the three problems described above. We describe the design and execution of a qualitative study of software explanation, the interpretation of results generated through qualitative analysis, and the exposition of theoretical constructs from our results in pattern form.

The grounded theory approach requires the researcher to clarify bias so the reader can determine if the account and results are reasonable. In Chapter 7, we describe the threats to validity. Then, in the concluding chapter of this thesis, Chapter 8, we provide a discussion of interpretations, implications and future directions while evaluating the respective contributions of Snapshot Theory, our process methodology and cognitive patterns.

In the next chapter, we will describe our work at greater depth through an exploration of the prior research in software comprehension and empirical software engineering.

[4] See glossary for definition of "pattern" and "pattern language".

# Chapter 2 Literature Review

## 2.1 Introduction

Software researchers study the activities of software professionals to achieve improvements to software process and software tools. Software explanation is a pivotal and observable yet poorly explored activity that combines speaking and sketching. The goal of software explanation is software understanding (either the explainer's understanding or that of his/her audience). Related empirical studies of software explanation may be found in the software-comprehension and software-diagram literature. Because of the paucity of earlier work on this topic, we need a research methodology that will allow us to generate a theory based on data from the ground up, i.e. a *grounded* theory. In order to address the challenge of constructing a grounded theory, we review methods for reporting our research. We will focus on the aspects of the reviewed topics that relate to the problems stated in the first chapter.

## 2.2 Empirical Software Engineering

Software researchers study the activities of software professionals to achieve improvements to software process and software tools.

In "The Role of Experimentation in Software Engineering" [17], Victor Basili suggests that progress in software engineering depends on our ability to understand the relationship between various process characteristics and product characteristics, e.g., what algorithms produce efficient solutions relevant to certain variables, what development processes produce what product characteristics and under what conditions. Basili states that, "models are built with good predictive capabilities based upon a deep understanding of the relationship between process and product." Scientific fields, e.g. physics, have progressed because of the interplay between two groups – theorists (who build models that predict events that can be measured) and experimentalists (who observe and measure). A recurring theme in many scientific fields is a pattern of modeling-experimenting-learning-remodelling.

One way to build a cognitive model[5] is through "software anthropology" field study techniques [79]. The application of field study techniques can result in the derivation of requirements for software tools, the improvement of software engineering work practices and the construction of new theories or hypotheses that can then be subjected to formal experimental validation. To determine typical behaviour, a researcher may "observe users, keep quiet, and let the users work as they normally would without interference" [99, pp. 18]. A second technique is simplified thinking aloud [99, pp.199-206], in which participants verbalize their thoughts while they perform a task. The verbal utterances allow the observer to determine *why* the participant is doing something in addition to *what* the participant is doing.

The talk-aloud (or think-aloud) protocol involves participants completing a task of solving a problem while they verbalise their thought processes [102]. In cognitive psychology, this technique has been used to evaluate comprehension processes in chess players and mathematicians. This technique has been used extensively to evaluate the cognitive processes employed by computer programmers; we review this in Section 2.4. Subjects are encouraged to speak freely while their verbalized thoughts are captured for subsequent analysis [83]. Ericsson and Simon [46] argue that, if the verbal data is produced concurrently and if the subjects' report on what enters their minds (rather than reporting explicitly on their comprehension processes) then the data produced is reliable. According to Russo *et al.* [116], talk-aloud verbal protocol provides the "richest source of a programmer's mental state."

In the past, researchers were criticized for basing their field studies on so-called "backyard research" [37]. Backyard research refers to studying students or novice programmers performing artificial tasks. There are two primary arguments against backyard research. First, since systems under study are small, the results are not scaled to adapt easily to large systems. Second, studying novices is inconsequential to industrial practice. Field studies with industrial participants in a natural setting are a reasonable way to address such issues.

[5] The term 'model' is overloaded in this dissertation: we refer to the model of user behaviour discovered through empirical research as a *cognitive model* (see glossary for our terminology).

There are two reasons a researcher might want to build a cognitive model to support tool development. First, tool developers can use cognitive models to understand those who will be using their tools. Secondly – and the tools community often neglects this point – by helping tool developers understand their users, cognitive models may actually improve the developers' own mental models. In the way that designers communicate through a user interface (a system image) to their users, users would ideally communicate with their designers through a "mental model image." This may complement the way in which users contribute to a system's design in requirements engineering [15] – a process not without problems [77, pp.5-13]. However, as users cannot communicate their "mental model images," software researchers produce cognitive models, and thus buffer communication between users and their designers. In this way, researchers contribute to the development of software design.

In our research, we perform the role of theorists who build cognitive models. The subject of our study of the activities of software professional is not the relationship between various process characteristics and product characteristics. Rather, we examine a process that developers regularly use – software explanation using a whiteboard[6] – and we analyse and convey the structure of this pivotal activity with the goal of extending software modeling tools to support this behaviour. In order to examine the discourse structure with which software professionals explain software, we ask professional software developers in a natural setting to solve a problem (how to explain software architecture to a new hire) while conveying their mental state through the richness of verbal utterances. Our object is to construct new theories or hypotheses that, in future work, can be subjected to formal experimental validation.

Researchers construct new theories using methods drawn from qualitative research, which we describe in the next section.

## 2.3 Qualitative Research

Scientific theories have often started with *qualitative research*: the study of behavioural regularities in everyday situations, for example, software explanation in

[6] Informal whiteboard sessions are a form of impromptu meeting in which software engineers gather to communicate about software through diagrams.

whiteboard sessions. Qualitative research involves the assembly, clustering, and organization of text, perceptions and social acts gathered through observation, interviews or documents in order to permit the contrasting, comparison, and analysis of patterns within them. A researcher expresses the results of qualitative research as "patterns" or "languages" or "rules."

Qualitative research involves three major operations: *description* (the context of study including participants' words), *analysis* (systematically identifying key factors and relationships), and *interpretation* (making sense of meaning in context). Harper [59] suggests that in addition to words, researchers can also acquire images as descriptive data that are still subject to interpretation.

As the computer science and software engineering fields have developed, there has been a trend towards quantitative research. Well-executed qualitative research [119, 120] will, however, always be important to generate new theories. Research can also mix aspects of both approaches (a.k.a. mixed methods research [33]) – for example, the collection of descriptive statistics about qualitative categories; such statistical analysis can be used to derive patterns from the data, which can then be subjected to further qualitative analysis. Two examples of statistical analysis include log-linear modeling (validation of hypotheses through estimation frequencies and comparison of frequencies with observed values) and Lag Sequential Analysis (LSA), which involves "determining whether or not the frequency of a given category is independent of the frequency of another category" [112]. LSA can be used to determine the random component of these patterns, helps to analyse the stability of the patterns, and requires less data than log-linear modeling to be significant. In our research, we restricted our inquiry to a qualitative approach without introducing quantitative analysis.

Qualitative research has many branches, including ethnography and grounded theory. In our research we, as a rule, followed the extended source book by Miles and Huberman, Qualitative Data Analysis [93].

The following are the features of qualitative research that tend to cross all branches of inquiry:

- There is prolonged contact with the field; multiple waves of data collection.

- The researcher is the main measurement device and is explicit about his or her role in the context under study.
- The research attempts to capture data on local actors "from the inside."
- The research isolates certain *themes* that should maintain their original form throughout the study.
- Multiple interpretations are possible, but some are more compelling for theoretical reasons.

The following are some of the analytic techniques typical of qualitative research. They are used across all branches of inquiry, but are most apparent in ethnography and are used in an extended form in grounded theory:

- Condense multiple data sources from observations or interviews by tagging categories of meaning using *codes*.
- Reduce data (e.g. focus or discard) such that "final" conclusions can be drawn and verified. This may involve the construction of tables or charts that help us to understand what is happening, how to analyse or take action based on that understanding.
- Transcribe remarks or rationale to codes.
- Sort or rearrange coded sequences to "identify similar phrases, relationships between variables, patterns, themes, distinct differences between subgroups, and common sequences."
- Bring patterns, processes, commonalities and differences back to the field for the next wave of data collection.
- Gradually construct a small set of generalizations that consistently appear in the data set.

Qualitative research in human-oriented sciences such as many aspects of software engineering will often be interactive and require active involvement by the participants. In such studies, the researcher gathers information (e.g. interviews, observations) while asking open-ended questions of the participants in a natural or realistic setting. Such a setting allows the researcher to gain deeper insight into the actual experiences of the participants. Sensitivity to the participant's time and effort is required; the researcher should not disturb

the 'site' more than necessary, and respect for the participants should be maintained throughout and beyond the life of the study. Qualitative research evolves as a study progresses (i.e. questions may be refined, data collection procedures may change).

We provide significantly more detail regarding the practical application of qualitative research methodologies in chapters 3 and 4. In the next section, we relate how our research builds upon previous qualitative research in the discovery of comprehension processes.

## 2.4 Comprehension Processes

By interacting with a product, an individual creates an understanding of that product in his or her mind. This understanding is the "mental model." The process of creating the mental model is called the "comprehension process"[7].

Researchers can apply qualitative research methods to study software comprehension[8]. One strategy is to explore a developer's mental model of a software system by having him or her present a detailed explanation of that system. Software explanation is a pivotal and observable yet poorly explored activity that links researchers to the cognitive patterns in their participants' minds. In the following, we will provide a review of the prior research, state the shortcomings of that research and describe how our research builds on that prior work.

The philosophical underpinnings of comprehension processes originate in cognitive psychology. The call for process-oriented studies of programming may be found in many sources [24, 36, 41, 57, 84]. An account of experimental methods to derive comprehension processes of computer programmers from software explanation may be found in the work of software researchers [13, 61, 80, 112], which we review below. These researchers reason that knowledge of programmer comprehension processes can lead to improved software process and software documentation. In particular, some researchers reason that since software maintenance tasks are challenging, we can improve maintenance capabilities if we understand and support the comprehension processes that developers use for over 50% of the maintenance task [127]. These improvements can become manifest in the maintenance

[7] Synonyms include cognitive processes and cognitive activities. See glossary for definitions of "mental model" and "comprehension process".

[8] See glossary for "comprehension (program / software)

process itself, in the design of documentation formats and guidelines and in the design of software tools. Even considering the sources we note, considerable work remains in the area of the comprehension processes of software developers.

Comprehension processes have been a rich source of theory and methods for twenty years and beneficial to the program-comprehension community. Meaningful practical applications based on the theories tend to be developed many years after the theoretical contribution. The gap between theory and application is wide in this research area. In the following, we examine the theoretical contributions in more detail.

The theory of comprehension processes in program comprehension originates in the work of Letovsky [80]. Letovsky studies comprehension processes at a meso-scale (tactics, on the order of seconds and minutes), as opposed to micro-scale (e.g. eye fixations, memory access) or macro-scale (strategies, on the order of minutes and hours). In the cited work, Letovsky collects think-aloud protocols from six professional programmers as they engage in a program-understanding task. Letovsky categorizes the 'understanding' portions (which are really *software explanation* portions) of the protocols analyses as:

- Inquiries, that is, questions and conjectures (the richest source of insight into the subject's thought processes, i.e. confusion, questioning, hypothesizing and concluding);
- Conversational exchanges with the interviewer (which reveal state of knowledge, but not cognitive events); and,
- Reading or scanning behaviour (easy understanding of the intentions of code).

Letovsky then analyses these explanation portions to provide evidence (sample data vignettes) for the following comprehension processes:

- Plausible slot filling (to integrate new code objects into prior expectations);
- Abduction (to hypothesize possible explanations for code objects);
- Planning (to hypothesize possible implementations for known goals);
- Symbolic evaluation (to determine what code does);
- Discourse rules (to guess the meaning of code based on meaningful names or coding style);
- Generic plans (to encode efficiency knowledge);

- Endorsement (level of confidence in assertion, that is, whether it is a guess or a conclusion).

Letovsky claims a complete mental model of a program should contain specification (the top layer: program goals), implementation (the bottom layer: actions and data structures), and annotation (an intermediary layer: the reasoning that links goals to actions and data structures).

Following Letovsky, Arunachalam and Sasso [13] study the comprehension processes six experts use when they perform software design recovery. The researchers produce a cognitive model of program comprehension in which they outline the following comprehension processes:

- Explanation generation (working hypothesis regarding presence, function, or structure of salient program component);
- Confirmatory association (additional features associated with working hypothesis);
- Explanation validation (determines validity of working hypothesis);
- Recording (written expression of explanation validation);
- Representation execution (transforms feature or recording into standardized symbolic representation);
- Synthesis (integration of distinct program elements with explanations of their purpose, mappings and representations).

In another study, Herbsleb *et al.* [61] study the patterns of interaction between the internal workings of individual minds and the rich artefacts (e.g. diagrams and prototypes) in the environment. Their work aims to assess the claims regarding the way in which object-oriented (OO) design is thought to enhance the functioning of software development teams. The research method is *ethnography*: in this study, researchers collect data from time sheets, videotapes of design meetings, meeting minutes, weekly interviews and general surveys. Aside from other analyses, the researchers perform an in-depth analysis of six videotapes of design meetings. Analysis proceeds according to a pre-defined coding scheme[9] of 22 categories of activity (the reliability behind the categories is described in

[9] We provide background information on coding schemes in Section 4.3 and we describe the coding scheme from our main study in Section 4.3.2.

more detail elsewhere [104], as is the coding scheme [62]). The researchers draw two relevant conclusions. First, adopting OO methods promotes effective communication, among team members, which requires less clarification. Second, OO design encourages a deeper inquiry into the reasoning underlying design decisions but less inquiry into requirements.

Robillard *et al.* [112] study the cognitive activities in team work. The goal of this study is to develop good practices by the comprehension of current software development activities, namely those found in technical review meetings. The researchers develop a coding scheme (not shown here). Then, they code the dialogues and mine the results to discover four categories of dialogue types, or cognitive activities, which they claim form the basis of technical review meetings. These four cognitive activities are:

- Review (cycles of evaluating-justifying activities);
- Conflict resolution (reject-evaluate-justify activities that indicate diverging opinions on criteria or solution evaluation);
- Alternative elaboration (sequences of development activities);
- Cognitive synchronization (request-inform-hypothesize activities that indicate participants share a common representation of design solutions or evaluation criteria).

The researchers suggest meetings dedicated to cognitive synchronization may improve the software engineering process. They further state that a thorough investigation of actual behaviours is required to better understand the intrinsic characteristics of meetings and provide the knowledge to assess current practices or render these practices more suitable to the practitioner's needs.

To this point, we have described how the study of comprehension processes has been a rich source of theory and methods. The four studies we have reviewed involve the development of cognitive theories from protocols in which study participants explain software. In the following, we will examine the relationship between cognition and explanation in more detail and consider further lines of research.

It is important to note that Letovsky was not "inside the programmer's mind" and could not identify cognitive events from the mind itself. He analysed fragments of software

explanation to find recurring behavioural patterns that provided clues about (not directly observable) internal cognitive events that indicated software understanding. He tagged data with explanation codes and built "crude theories of the mental representations and processes that produce questions and conjectures." [80]

Comprehension process research, of the type we present in this thesis, typically follows Letovsky's method of discovering comprehension processes based on software explanation. Doing so relies on two assumptions. The first assumption is that in the task assigned, the programmer manipulates their mental model. The second assumption is that verbal protocols of professional programmers explaining software provide a close approximation to "a trace on the subjects' thought processes." [80] We will now examine the ways in which we can improve upon the prior research.

Let us investigate the first assumption, that the task that is set for participants allow them to manipulate their mental model[10]. Letovsky sets a task for participants: to plan a modification to a program and think aloud while they interpreted print-outs of source code. Arunchalam and Sasso set a task for participants: to think aloud while they studied an assembler program until they understood it well enough to explain its function and implementation to another programmer. Herbsleb *et al.* did not set explicit tasks: the researchers recorded and analysed actual design meetings with multiple team members over a long period of time. Similarly, Robillard *et al.* recorded and analysed team environments in the context of technical review meetings.

Our research is more in the spirit of the research of Letovsky and Arunchalam and Sasso: we interviewed a single software developer at a time, rather than a team, and because we asked explicit questions, we had a higher degree of control over the study than the studies of Herbsleb *et al.* and Robillard *et al.* Letovsky identified participant inquiries as "the richest source of insight into the subject's thought processes." We extend this line of reasoning in our study: by structuring our interview to regularly prompt inquiry moments we should receive the richest response.

O'Brien [103] criticized cognitive studies in which researchers carry out "tightly controlled experiments in an artificial environment" and argued for the observation of real

[10] The prior work does not make the explicit point that the goal of software explanation is software understanding.

activities in real situations. Unlike the research of Letovsky and Arunchalam and Sasso, we investigated a task that is consistent with the software developer's daily agenda, regular communication activities and natural environment.

In each of the prior studies, the basis for the task allowing the participant to manipulate their mental model is that participants perceive and interact with external stimuli, access short- and long-term memory and use external memory to store and organize explanation information. However, it is highly doubtful that this basis for mental-model manipulation can be determined by the protocol alone. This leads us to the second assumption, that protocol analysis provides the closest approximation of programmers' thoughts. This assumption is widely received but contains the flaw that software explanation is enhanced through speaking *and* sketching. Software explanation may be possible without sketching, but in our research we found that sketching was an intrinsic facet of explanation. An analysis of software explanation should therefore simultaneously address both speaking and sketching activities. One source that supports this theory is the "Mind's Eye Hypothesis," which states that "there is a close relation between the kinds of diagrams people draw on paper and their mental images." [109] Because we study the protocol in addition to the diagrams, we enhance the prior research and not only get a closer approximation of our participants' comprehension processes, but are also more likely to find events that accurately represent the participant's manipulation of their mental model.

Arunchalam and Sasso set a goal to "uncover cognitive patterns in protocol data that reflect program comprehension," a goal that characterizes the state of comprehension-process research. In our approach, we too seek to uncover cognitive patterns, but we make the critical distinction that software explanation constitutes both speaking and sketching. In the following section, we support the study of the sketching side of explanation through an examination of why empirical research of diagrams in software engineering is timely and relevant.

## 2.5 The sketching side of software explanation

Empirical research in software diagrams is timely and relevant because software engineers increasingly depend on software tools to support graphic representations (e.g. UML [3]), but such tools may not be grounded in the way software engineers actually use

diagrammatic representations. Cognitive analysis of UML is an interesting and beneficial research area of diagrammatic reasoning[11] that merits investigation [40]. In this section, we justify the examination of diagrams in software engineering.

There is evidence that diagram and visualization tools have been notably ineffective to date [130]. Software engineering is becoming increasingly model-driven[12] [4, 42, 65]. The purpose of a software model is to provide an abstract representation of a system that allows humans to deal with software complexity by reasoning and communicating about smaller, modularized representations. It follows that tools that support software models must cull software complexity by supporting the reasoning and ability to communicate modularized representations. Diagrammatic reasoning has been of interest for centuries and the last decade has seen a sharp increase in the computational and cognitive perspectives of diagrams (e.g. [56, 58, 97]).

We observed that many software engineers in our fieldwork avoid using representational tools. When asked to describe their system they use a variety of approaches not necessarily found in UML. Software engineering tools tend to be awkward to use. Tool developers do not currently have the knowledge of what makes tools easy to use that would allow them to develop more effective tools.

Why are diagrams useful? Larkin and Simon [75] describe a theory (later confirmed by Cheng [29]) of cognitive processes involving manipulation of spatial structures to help scientists control the search for a solution to a problem. The importance of this widely-cited work is that graphic representations such as diagrams are useful because they are analogous to the world they represent and therefore do the 'heavy lifting' in the recognition process.

The study by Butcher and Kintsch [27] addresses why diagrams improve memory and learning. The goal of the study was to determine if the comprehension processes of learners who use text and diagrams differs from learners who use text only, and whether diagram complexity influences comprehension processes. In pursuit of this goal, the researchers

[11] See glossary for definition of "diagrammatic reasoning"

[12] Thus, we assume software engineers will use, should use, or do currently use modeling languages (e.g. UML); however, these assumptions require empirical evidence.

apply protocol analysis to develop a coding scheme which may be useful for other researchers who empirically study diagrams. They score propositions as paraphrases (those that reflect information from current materials), elaborations (connections to prior knowledge), monitoring statements (comprehension monitoring), or self-explanation inferences (synthesizing or integrating materials). The researchers find diagrams effective when they induce learners to employ the comprehension processes necessary for deeper understanding.

Suwa and Tversky [132] address the reasons designers draw sketches. The authors indicate that external representations relieve memory load in two ways: they provide external tokens for the elements that must otherwise be kept in mind, and they serve as visual-spatial retrieval cues. The authors conjecture the reason designers use sketches is as follows:

> In developing ideas for new projects, designers do not draw sketches to externally represent ideas that are already consolidated in their minds. Rather, they draw sketches to try out new ideas, usually vague and uncertain ones. By examining the externalizations, designers can spot problems they may not have anticipated. More than that, they can see new features and relations among elements that they have drawn, ones not intended in the original sketch. These unintended discoveries promote new ideas and refine current ones. This process is iterative as design progresses.

This quote underlines the central role of the sketch in iterative design. Designers build external representations to support the design process. Designers continually rebuild external representations as development proceeds. With these two postulates, we propose that the *temporal details* of diagrams are important. Ader *et al.* [6] also suggest that temporal details may generate new research hypotheses.

A consideration of the temporal details of diagrams may offer a clue to practitioner's general apathy towards software diagrams. The weakness of diagrams may be that although they offer deep insight, such insight is often for only a limited period of time. James Cordy informally stated during a workshop discussion [2], "Once you have absorbed a diagram's content, two minutes later, it may not be of interest again for the rest of your life." This opinion reminds the reader that the temporal quality exists, though it may exist in a

transitory state, and leads us to consider the temporal details of software explanation as the central phenomenon in our empirical study.

Software researchers investigating comprehension processes have not yet studied the sketching aspect of software explanation. A study of the temporal details of software explanation is novel and we need a qualitative research methodology that will allow us to generate a theory from the ground up: a "grounded theory."

## 2.6 Building a theory of software explanation from the ground up

Grounded theory research [55, 93, 131] is a rigorous qualitative research methodology in which the researcher develops *theories* that are 'grounded' in observational data, in particular the observation of people. Several software researchers have already developed theories by using grounded theory methods [28, 107, 123]. We need such theories in software engineering (c.f. Schank's Theory of Natural Language Understanding [121]), since a deep evidence-based understanding of human activities can lead to the development of better tools and processes. Historically, development of such tools and processes has been primarily based on "folk knowledge" – beliefs that lack scientific validation [140, pp.21]. We adopt the key tenets of grounded theory in Chapter 3 and Chapter 4.

Some of the key tenets of grounded theory are:

- Researchers should generate open-ended data by asking open-ended questions.
- Researchers should perform *theoretical sampling*. This means the selection of study participants from a wide variety of groups or people[13] – in our case, different people who are involved in software comprehension.
- Researchers should avoid preconceptions regarding theoretical outcomes. More specifically, they should avoid biased interpretations of the data, interpretations derived from concepts in the literature. The theory must emerge from the data. Typically, grounded theory researchers perform a literature review only when the process of making assumptions and interpretation of data is underway or complete.

[13] We sampled two groups. Within these groups, we achieved variety by sampling both experts and novices, and people with specific domain knowledge. Our research, following similar work described Section 2.4, has a reasonable variety of individuals for demonstrating the process.

- Researchers must analyse data using an empirical, repetitive and on-going process that involves continual reflection. They ask analytical questions, organize the information into coded categories, adjust existing categorizations and record observations and information that will aid further analysis. This standard grounded theory approach is called the *constant comparative* approach.
- Researchers must focus on a single or central phenomenon regardless of the widely varying and interesting details the data may present.
- In sociology, researchers report their results as lengthy monologues. In our approach, we instead produce a theory for the consumption of academics and a set of shorter cognitive patterns with an emphasis on application.

Grounded theory may be valid for theory building in software engineering, but the use of these methods involves some significant challenges [14, 33]. We will highlight the challenges in later chapters, but an interesting challenge to the novice researcher is the inductive literature review. To reinforce the emergent properties of grounded theory, the researcher begins research with only a topic and avoids bias from prior studies by the omission of the literature. This is meant to promote unbiased theory building, but in practice it is a significant challenge for the (novice) researcher. For practical purposes, the scientist should combine rigorous grounded theory techniques with perspectives drawn from the literature.

The second issue that we face is that protocol analysis and grounded theory are both inadequate for our purposes. Protocol analysis is too specific to the natural language of our study participants and does not support coding based on software representations. Grounded theory is too broad and would result in book-length texts surrounding our inquiry. We do however have an opportunity to produce a research methodology that combines and extends empirical elements of both techniques. This method, described in Chapter 3, allows us to examine the first research question.

The third, critically important problem we face in using grounded theory research is how to report our results in a form that is scientifically valid and accessible to readers. We address this challenge in the next section.

## 2.7 Reporting the results of empirical research in pattern form

Arunachalam and Sasso propose that the description behind comprehension processes can eventually contribute to the development of computer-based support for program comprehension – more specifically, software support tools. The authors produce a scholarly review, worthy analysis and interesting results, but defer recommendations for the development of software support tools to future work (although they state that these recommendations are a primary objective of the study). Such deferment is a common weakness in the comprehension-process research we reviewed: strong in theory and method, they are often weak in the application of results. In an effort to methodically build upon the prior research, we faced similar challenges in our research. To address the gap between theory and application[14] we constructed two versions of our theory, one for the consumption of academics and one for the consumption of tool designers. We describe these versions in later chapters. We draft the latter in *pattern form*, as Martin *et al.* [88] recommends, which provides a focus on structure and consumption by a specific audience.

Patterns are a phenomenon that has become entrenched in software engineering since the mid-1990s. Patterns focus on a deep understanding of recurring problems. Each pattern addresses an individual problem, though sometimes patterns work together to solve bigger problems. Pattern authors document patterns in discourse, a process that provides an informal catalogue of knowledge. Pattern writing is notoriously difficult. The pattern community, aware of this, provides a number of sources to aid in the writing process [92, 111, 137, 138]. The pattern community reasons that, by recording solutions to recurring problems for reference, individuals can focus on larger, tougher challenges. Without patterns, software developers re-develop rather than re-use. The general consensus is that software is complex enough, and patterns may reduce this complexity.

In the late 1970s, Christopher Alexander [8, 9], concerned with the state of his field of urban design and building architecture, strove to assemble an understanding of the structural artefacts that shape daily interactions between people, with an emphasis not only on the structure of buildings, but also on a structure of the natural social groups in

[14] As future work, members of our research group are building instantiations of our theory in an IBM software development tool.

buildings [31]. He coined the terms 'pattern', 'pattern language' and 'forces' to serve his vision of elucidating knowledge of patterns that make people feel "fully alive" within his architectural creations. We associate the early value system (what makes a pattern 'good'), and the way in which we write patterns, with Christopher Alexander. From Alexander [8], a pattern is a 'good' one if you can visualize the solution it generates concretely, and if the pattern's inner forces resolve themselves. We append the following: the problem must be relevant to the audience, and the pattern must exhibit clear writing.

Patterns are becoming widely used in software engineering and other fields as a way of capturing expertise within a community of experts. The best known patterns are those of the so-called "Gang of Four", Gamma, Helm, Johnson and Vlissides [52]. These patterns are called design patterns since they represent common approaches to solving object-oriented (OO) design problems. In addition, there is a growing wealth of other types of patterns, noted in the Pattern Languages of Program Design books (e.g. [89]), including work similar to ours in reengineering patterns [39] and usability patterns. "Activity patterns" are another area of pattern interest. These are recurring properties of human activity that do not necessarily pass judgment on whether the observed activity pattern is good or bad [134].

Gardner *et al.* [53] set the precedent for applying cognition in the realm of OO technology. Their approach is best described as the application of cognitive models to organizational and system processes, and is highly concrete. Cognitive patterns (in their words, templates for how humans solve problems) serve as a framework for OO projects. Although we share similar roots for our research, a fundamental difference exists: their work is closer to business process modelling and less about patterns in the Alexandrian sense.

We base the transition from grounded theory to pattern writing on a near perfect fit between the two approaches:

- Both approaches are focused on a deep understanding of a central topic.
- In grounded theory, the phenomenon surrounds a set of themes or categories. In the pattern world, these are the patterns themselves, and the names of the patterns are analogous to the names of the categories.

- The source of both categories and patterns are linked to recurring themes in the real world.
- Patterns focus on a deep understanding of a central problem. This is not unlike the focus in grounded theory on a central phenomenon.
- In grounded theory, one attempts to understand the contextual and primary conditions that allow the phenomenon to exist. In the pattern world, the context and forces are two of the most important features of patterns. The concept of forces extends beyond primary conditions.
- In the pattern world, we find solutions and rationales as to why that solution is appropriate. In grounded theory, we find the outcome of phenomenon and the strategies leading to those outcomes.
- Both grounded theory and patterns concentrate on consequences.
- In grounded theory, we seek to understand common threads and differences among categories. In the pattern world, we find the analogous concept of hierarchy and linkages.
- Patterns are written documentation, and many agree the drafting of patterns is difficult. Likewise, the output of grounded theory is a narrative, and few dispute the difficulty of the approach.
- The concept of a visual model in grounded theory aligns with the concept of dynamics and structure in the pattern world. If the phenomenon in question is a problem, a researcher may draft the output of grounded theory in pattern form.

We see a strong continuum between grounded theory and patterns – grounded theory focuses on building theory, and patterns offer the medium for presenting theory. The upshot of this finding is our realization that the goals of our research are in the spirit intended by Alexander.

## 2.8 Concluding Remarks

In this chapter, with the goal of providing a convincing structure for our research, we reviewed the literature that most closely influenced the research questions stated in the first chapter.

We perform the role of theorists who build cognitive models with the goal of extending software modeling tools to support behavioural traits of software explanation. We recognize the crucial research step that will allow the extension of software modeling tools is the construction of a research methodology and generation of a theory. We came to this realisation from an examination of prior research in comprehension processes.

Some of this prior research examined programmers working with source code, while other research examined communication involved in different kinds of meetings. We noted the methodological deficiencies that we can improve with our approach. The most notable deficiency was the prior research's failure to examine representations of software, e.g. UML or freeform diagrams, as a common medium for software explanation. This drew our attention to prior empirical studies of diagrams in software engineering. We noted the novelty offered by a qualitative study of software explanation employing both speech and sketch activities. We then reviewed a research methodology that would allow us to generate a theory from the ground up – a grounded theory. In so doing, we encountered the issue of how to report our research, an issue that is resolved by writing our theory in pattern form as cognitive patterns. This allows us to make a contribution to software design: tool developers may use cognitive patterns to understand their users and cognitive patterns may actually improve the tool developers' own mental models.

In the following chapters, we describe the methods and results of our research approach.

# Chapter 3 Pilot Study

This chapter describes a pilot study, the purpose of which is to identify a behavioural phenomenon in software professionals and to devise categories of software explanation based on the observation of such professionals working in the field. These categories or codes will subsequently receive deeper examination in the main study. The goal of the main study is to generate a theoretical foundation for a more detailed investigation of the existence and importance of the snapshot phenomenon in the creation of diagrams, and more generally in the process of software explanation. Our strategy involves videotaping whiteboard activities while professional software engineers explain software, coding the video data, defining topical categories from the codes, and exploring trends in the data through the use of visual aids, or *displays*, such as tables, figures and other models that illustrate interesting aspects of data. In this chapter, we describe the initial steps of this strategy.

## 3.1 Objective

The objective of the pilot study was to develop a preliminary set of codes to be corroborated in the main study. At the pilot stage, we did not seek to build a theory or model. Our aim was, rather, to familiarize the principal researcher with the coding process, to identify and overcome challenges that might hinder data capture and data analysis, and to identify and develop the core set of codes and categories that we would examine in greater depth in the main study. We also provide a discussion of some of the underlying principles in the formation and execution of a qualitative inquiry.

In the early stages of this research, our research objectives had not yet been fine-tuned and our research method had not yet been worked out. After the pilot study and initial analysis, we decided that grounded theory was appropriate for the analysis of our codes. The decision to use grounded theory was followed by the formulation of our second major research objective: to develop a methodology in order to study the discourse structure of software developers. This chapter illustrates the formative stage of this decision.

## 3.2 Participants and Study Setting

The setting for the pilot study was the on-going maintenance of a complex real-time telecommunications software system at Mitel Corporation. A team of developers regularly update the telecommunications software: their routine and extended experience with whiteboard sessions made them ideal candidates for our study. Unfortunately, participant selection could not be random due to the relatively small number of people available. We felt fortunate that twelve Mitel technical leads and developers were kind enough to participate in the project. With their involvement, the principal researcher held twelve videotaped informal whiteboard sessions on location over a six-month period in 2002.

This pilot study focuses on the informal whiteboard session process. We initially relied on our own personal experiences with informal whiteboard sessions. Throughout the study, however, we updated our understanding of the informal whiteboard process through observation and interviews. This study was not intended to evaluate what was being written or drawn on the whiteboard, but instead to understand the reasons and ways a developer uses the whiteboard.

Generally speaking, informal whiteboard sessions are held at varied and random times throughout maintenance and development lifecycles. Often developers need to draw on the knowledge of their peers, need to work through problems using their peers as a sounding board, or just need to step away from the computer and think using visual representations. Informal whiteboard sessions involve one to many developers with varying degrees of expertise and knowledge related to the subject of the sessions.

## 3.3 Data Collection Procedure

In this pilot study, we combined interview questions with the taping of a think-aloud protocol on videotape. We used the same data collection procedure in the main study.

During the pilot study, we captured data at Mitel using an analog video camera and converted the data to a digital format using video capture hardware and software. During the main study, we captured data with a digital video camera on-site in meeting rooms, and imported the data to computer using video capture software that produced higher quality video. Video captures a complete record of a session, which makes it both detail-rich and

time-intensive to use and interpret. The rule of ten, that every minute of captured video requires ten minutes to process, may constitute an underestimate if the analytic method is too open-ended; in our study, we analysed roughly twenty minutes for every minute of data. Each session lasted 30-45 minutes and involved a single participant. We used both the participant's verbal utterances and whiteboard diagrams as the basic data for our analysis.

We prompted the participants by asking each one several questions. According to our data, the following questions generated the most diagrammatic material:

- Imagine I am a new hire with basic knowledge of computer science, telephony and real-time systems; please explain the architecture of the [system you maintain] and related systems to me.
- Please explain very briefly the main subsystems, layers, processes and data structures.
- Please explain how the architecture supports the following features.

The challenges inherent in using interview sessions as the data collection method for a qualitative inquiry include:

- how to structure questions;
- how to prompt interviewees without leading;
- how to run the session and simultaneously record notes;
- how to properly manage interview time.

To address these challenges, we prepared an investigator's handbook as part of the full set of study materials[15].

## 3.4 Data Analysis Procedure

After the interview sessions, we transcribed all tapes as the first step in our analysis. Our progress was somewhat hindered at this stage because our analytic procedure was not rigorous; eventually, we evolved our analytic procedure to match the process illustrated in Figure 4.1. We reviewed the video data to make sense of the whole and to find regularities. We were not yet sensitive to the subtleties of the patterns we found.

[15] A full set of study materials may be found online at http://www.site.uottawa.ca/~tcl/gradtheses/amurray/appendices/.

The following describes how we identified at this initial stage a plausible phenomenon to examine at greater depth. During an interview session at Mitel, our participant provided particularly rich details for nearly forty minutes. At the whiteboard, the participant regularly remembered new details that he wanted to add to his story of how a practical phone call takes place in terms of PBX technology; his way of adding the new details was startling. He would find the space on the board to add the detail, would announce the level of importance of the upcoming information, and through a series of added and removed details would animate the process in conjunction with his verbal annotation – his diagram was constantly changing, and when the session was complete, the diagram did not reveal any of the dynamic story-telling that had taken place. This case drew our attention to the novel yet simple idea that temporal details may be important in software explanation.

The concept of temporal details looked reasonable, sensible and interesting, so we sought a deeper understanding through qualitative data analysis. This became our analysis of discourse structure. In pursuit of this analysis, the observed behavioural patterns became our initial coding scheme, which we present in the next section.

## 3.5 Results: Coding Scheme

The raw data from our pilot study consisted of twelve videotaped interviews, or 7:54:48[16] of raw video. When we removed extraneous details (e.g. camera set–up, meeting and greeting participants, etc.), we were left with 6:44:55 of processed interview content. Percentages of total time given in this section are relative to the processed interview content. In total, eighty-two scripted questions were asked of participants, or an average of 6.83 of a possible 10 questions per participant (see Section 3.3 for examples of questions).

Several passes of the data led us to a scheme with 13 distinct codes. These can be roughly grouped into four categories we call "drawing", "speaking", "snapshot", and "other." "Other" codes corresponded to thinking activities and were the hardest to identify, though we were able to recognize recurring patterns of cognition manifested through both speaking and drawing. In the main study, we abandoned the "other codes" because the trail of evidence was too difficult to construct and defend. Likewise, many other codes were not

[16] i.e. seven hours, fifty-four minutes, forty-eight seconds

further developed in the main study because they did not contribute to the central phenomenon under investigation.

Participants spent 57% (3:48:54) of their time speaking and 25% (1:43:01) of their time drawing. A total of 18% of the session time was allocated to noise; by noise we mean time when the interviewer was asking questions, interruptions, and spaces longer than five seconds where neither drawing nor speaking occurred.

We use "speaking" to refer to the time during which participants were verbalizing but not drawing. When the participant drew on the whiteboard we exclusively used drawing codes, and if the participant was speaking while not drawing, we used speaking codes.

The participant was considered to be "drawing" whenever he or she was moving a marker on the whiteboard; this was recorded to the second. The fact that 25% of total time was spent drawing shows that we gathered much potentially useful information about diagrammatic drawing. While "drawing", participants exhibited different forms of behaviour including: silence, talking about diagrams, confirming aloud their thoughts, random discussion, and joking. In other words, participants exhibited natural and typical whiteboard activity.

We found other results, but they do not contribute to the central thesis, nor do they contribute to the central phenomenon under study. For these reasons, we will not present them here. Figure 3.1 illustrates our original coding scheme.

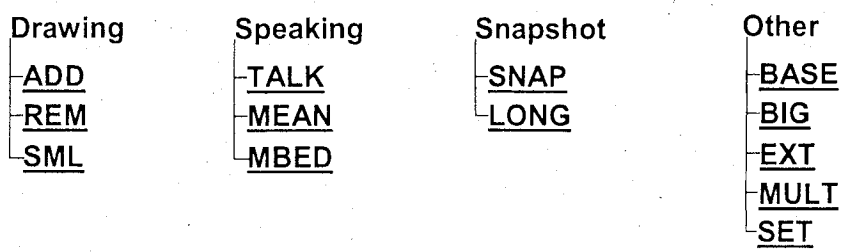


Figure 3.1: Pilot Coding Scheme

**ADD-Add Detail**: The participant draws new material on whiteboard.

**REM-Remove Detail**: The participant *modifies previous material*, or *erases* some or all of the material on whiteboard.

**SML-Starting Small**: The participant *begins a new representation.*

**TALK-Talking**: The participant is speaking, neither drawing nor discussing prior nor future diagrams.

**MEAN-Retain Meaning**: The participant references other diagram or diagrammatic elements and *describes meaning not captured through the diagram itself.*

**MBED-Embedded Rationale**: The participant describes his or her rationale for drawing.

**SNAP-Snapshot**: *A representation in a moment of time.* A representation contains enough information to allow the participant to speak to some meaning. Often the participant pauses to reaffirm the details contained within a representation.

**LONG-Longview**: This is a *series of snapshots*, typified by (though not exclusive to) steps leading to the completion of a diagram.

**BASE-Baseline Landmark**: A participant continually references, either verbally or via representation, a *central or familiar concept.*

**BIG-Thinking Big**: The participant describes some unifying concept using a complete representation as opposed to its constituent parts.

**EXT-External Memory**: The participant uses external media in order to store details for later recall.

**MULT-Multiple Approaches**: Different participants typically *represent the same concept in different ways.*

**SET-Set Boundaries**: This code marks places in which participants *set boundaries when answering the questions.*

## 3.6 Concluding Remarks

The pilot study was important. We developed a preliminary set of codes, which we examined in greater depth in the main study. We gained sufficient experience in the coding process to allow us to competently perform in the main study. We also identified several

challenges in data capture and analysis. We document how to overcome these challenges in the next chapter.

# Chapter 4 Study Design

## 4.1 Introduction

Our qualitative pilot study provided a systematic way of organizing and understanding the data through a set of codes. The initial coding scheme remained to be tested and cross-verified through the use of an independent sample. As a consequence of performing our second qualitative study with an independent sample, the coding scheme – and therefore the analyses we derive from the coding scheme – became more complete. In general, a qualitative study will result in propositions tied to a trail of evidence, hypotheses grounded in the data, explanations regarding the phenomenon under study, and areas for future study. In this chapter, we describe how we designed our study to achieve these results in practice.

The literature offers several well-known strategies [25, 80, 85, 122, 139] that describe how people approach comprehension problems at a high level. We however are interested in the way people manipulate their mental model at a somewhat lower level: from minute to minute or even second to second as they tackle a comprehension or explanation problem. This is similar to the low-level comprehension processes, or cognitive activities, we reviewed in Section 2.4.

We introduce this chapter by describing the impetus for the study and research objectives. In Section 4.2.1, we describe our participants and the study setting. In Section 4.2.2, we provide our study design. In Section 4.3, we detail our data analysis procedure, which includes our revised coding scheme and our motivation and mechanism for data displays (i.e. tables and charts from which we draw conclusions). The aim of this chapter is to provide the overall design for our study. We describe the results of our study in the next chapter.

### 4.1.1 Impetus for the study

From a theoretical perspective, a behavioural analysis of software explanation appears likely to improve our understanding of how software developers systematize and express their knowledge structures when explaining software (i.e., speaking and drawing). From an applied perspective, a comprehensive description of the temporal process of software

explanation may shed light on how software development tools can better support the creation of software models.

This chapter presents three contributions made by this thesis, all of which relate to our second research problem: which methodology to use. The first and most important is a demonstration of qualitative research methods incorporating concepts of the *grounded theory* method, discussed in Chapter 2. In the previous chapter, we built the foundation of our work: the development of an initial set of codes. In this chapter, we demonstrate how qualitative research is conducted based on the tenets of grounded theory and a detailed set of analytic steps. In particular, the range of displays built in conjunction with a rigorous analytic process may yield new insight into difficult software engineering problems. We use the grounded theory approach for two reasons: first, we are studying the process of human behaviour; and second, there is a lack of earlier work along the lines of our research. This means that we do not yet have any hypotheses on which to base more quantitative studies. We believe this grounded theory approach has led to results that are rich, descriptive and closely linked to the data.

The second, related, contribution developed in this chapter is the refinement of a protocol, piloted in the last chapter, for examining *informal whiteboard sessions* in which a software engineer explains software.

The third contribution described in the chapter is the development of a more refined set of codes that mark behavioural activities performed during the whiteboard sessions.

### 4.1.2 Objectives

The primary objective of this study is to clarify the role of temporal details as an experienced programmer explains his or her knowledge of structure and functionality of a software system. In particular, we will explore the *snapshot* as the foundation of temporal details, and will answer the research questions we posed in Section 1.1.

Our objective is to generate a *Snapshot Theory*. We provide a systematic description of the software explanation process supplemented with evidence that supports the snapshot as the participant's building block towards software understanding. As part of this inquiry, we progress towards the secondary objective: to develop a methodology in order to study the discourse structure of software developers.

## 4.2 Method

### 4.2.1 Participants and Study Setting

Twelve IBM developers participated in the study. Participants had completed at least an undergraduate degree. Two participants were developers for the Eclipse framework, and the other ten were developers of IBM Rational Software Architect. Participants worked in a variety of software development roles and on various product components. Participant selection was not random. Their level of software development experience ranged from four to twenty-four years; their level of experience with the product in question ranged from six months to seven years; and their use of informal whiteboard sessions was typically two to five times per week.

Prior to recruiting the participants, we attained approval from the University of Ottawa Ethics Review Board for the entire study.

To recruit participants, we used the following process:

- We obtained management support from IBM.
- We sent a recruitment letter via email to IBM managers and developers (see online Appendix).
- We met individual developers, described the study in more detail, and when a developer volunteered, allocated specific time and place.
- We sent an email reminder and thanks to each participant.
- If the participant was late for a session, we followed up with a desk visit (participants sometimes required a final reminder).

### 4.2.2 Study Design

We designed the study in order to examine the behaviour and patterns of experienced software developers engaged in software explanation during informal whiteboard sessions. In a closed environment (i.e. with the participant alone in a meeting space with the principal investigator), we asked the software developers to explain their mental models of the architecture of the software they were developing. We asked participants to think aloud as they drew diagrams.

A typical session was run as follows:

- Investigator welcomes participant.
- The investigator informs participant:
  - of the study objective;
  - that participation is voluntary, and that the participant has a right to terminate the session at any time;
  - that investigators have management support, but that managers will not see the data;
  - that investigators will preserve anonymity and confidentiality of the participant, of the session, of the results, and of the system under study;
  - that the investigators are interested in observing the way participants explain software to new hires; and
  - that investigators are studying what the participant says and draws on the whiteboard, and are not studying the participant personally.
- The participant reads and signs the informed consent form.
- The investigator introduces the study procedure, including proposed timeline and study equipment.
- The investigator invites the participant to ask questions at any time during the session.
- The investigator starts taping.
- The investigator asks a series of interview questions (see online Appendix).
- The investigator reminds the participant to use the whiteboard and to verbalize their thoughts, in other words to "think aloud."
- The investigator takes notes, and prompts the participant for further details where appropriate.
- The participant answers interview questions as they see fit.
- If the study time elapses or the investigator has no further questions, then the investigator requests clarification of any details he is unsure about, and asks the participant if he or she wishes to add anything else.
- The investigator stops taping.

- The participant fills out a follow-up questionnaire regarding details of their level of experience in IT, the product with which they work, and UML diagrams. We discuss this later in the thesis.
- The investigator thanks the participant for their time and patience.

We counter the Hawthorne effect[17] by asking participants to consider the study analogous to an informal session in which they explain software to a new employee.

Not every participant provided an answer to every question, though incomplete question coverage did not really matter. The purpose of the questions was more to ensure that we gathered a wide variety of data. We were interested in studying what was said and drawn, not the specific details of the answers themselves. For a complete description of the questions asked in the whiteboard sessions, please see the online Appendix.

### 4.2.3 Data Collection Procedure

We used the same data collection technique described in the pilot study, Section 3.3 with the slight difference that we tailored the questions to the product with which the participants were familiar.

## 4.3 Data Analysis

We followed the *constant comparative method* as dictated by the grounded theory approach presented in Chapter 2. Grounded theory is largely based on content analysis. In our research, we focus on behavioural analysis (not content analysis), and, thus, we follow the tenets of grounded theory as part of a more general qualitative inquiry. In accord with a grounded theory approach, we coded and categorized the data, grounded observations in the data without preconception, made propositions based on these observations, and verified these propositions in the data.

The richness of the video content had the potential to lead us to a state of 'analysis paralysis.' As investigators, we had many options for coding. We found that following the tenets of grounded theory, such as *focusing on a single central phenomenon*, was of considerable help in allowing us to pursue our research objectives; we discuss this in more

[17] The Hawthorne effect refers to a participant's altered behaviour produced by obtrusive observation -- in other words, the scientist must attempt to ensure that the study conditions do not alter participant behaviour.

detail below. In the pilot study, we moved from video to codes and categories through the application of the first three major steps in the process model, illustrated in Figure 4.1. As part of the main study only, we applied all five major steps of the process model.

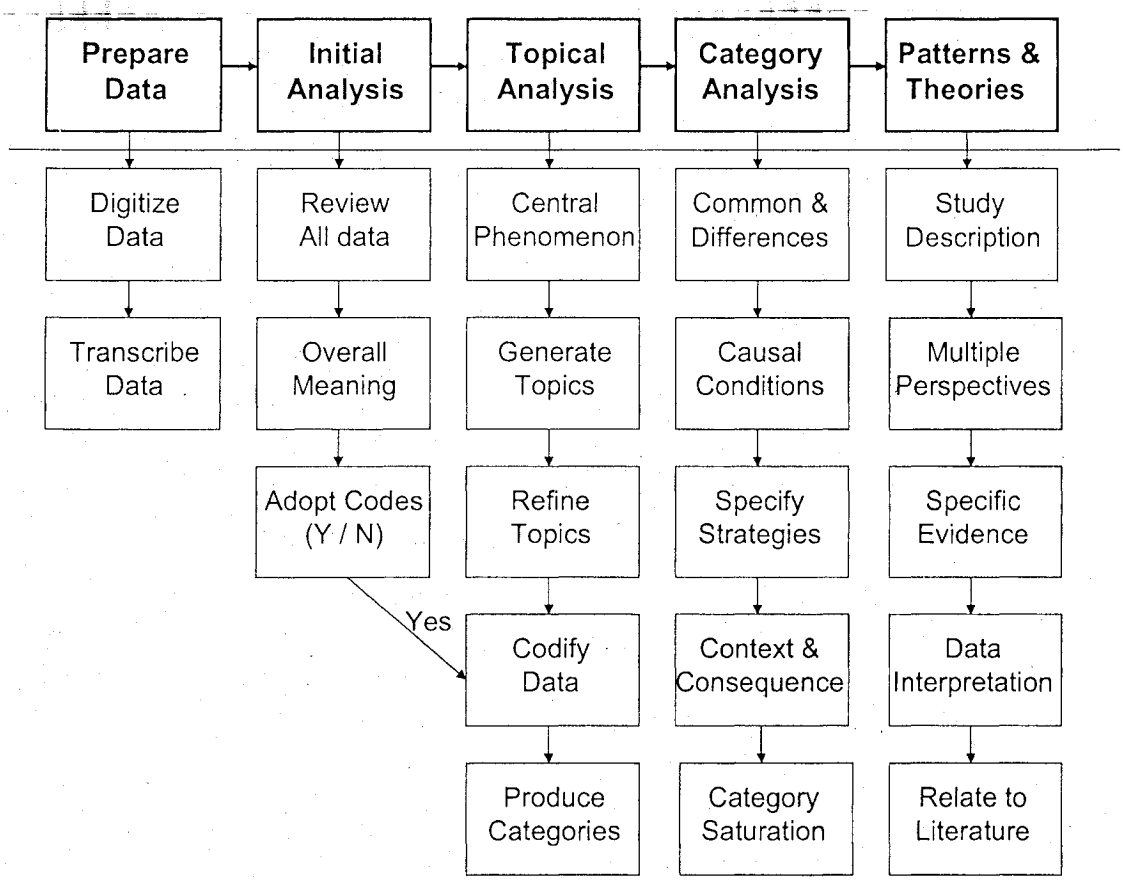


Figure 4.1: Qualitative Process Model: Moving from video data to codes and categories

We describe below how the process model shaped our work.

**Central phenomenon**

A central phenomenon is some particularly interesting aspect of the data. We began by picking a videotaped session and reflected on its underlying meaning. The central phenomenon we identified was the existence and importance of snapshots in the creation of diagrams, and more generally in the process of software explanation.

### Topics and categories

If a researcher keeps the central phenomenon under constant consideration, a review of the data from other participants yields different manifestations of the phenomenon. These sometimes required us to expand the categories of observations and group them into high-level categories. For example, two of the high-level categories in our analysis were 'speaking' and 'drawing.' We also went back to data we had already analysed to ensure we properly recorded the information about each newly added category. In addition to forming new categories, we grouped together those that were similar. The process of ensuring that all the information relevant to the central phenomenon is captured with an adequate set of categories is called *category saturation*.

In the grounded theory approach, the simple codes used to mark the categories of data are usually called 'topics,' whereas a description of the category of data is called a 'category.' We will, however, use the term 'codes' for simplicity here. Codes ease the identification of trends, patterns, and generalities and can help extract quantitative data from qualitative data.

### To adopt a prior coding scheme

A researcher may use a coding scheme from prior research. We debated using one of the published coding schemes [13, 61, 81, 112]; however, the graphical nature of whiteboard work confirmed the need for a unique set of codes. The other schemes did not fit the context of informal whiteboard sessions. In general, it may be viable to create a new coding scheme where one has not previously existed, but one must exercise diligence in examining the possibilities. As Figure 4.1 shows, adopting a coding scheme contributes to efficiency.

### To devise a new coding scheme

Literature about the grounded theory approach [23] supports our finding that a researcher charged with devising a new coding scheme may encounter numerous issues. These include deciding which codes are *relevant* and will lead to answers to research questions. In our research, we had to drop many of our initial leads[18] that were irrelevant to

[18] C.f. pilot study codes and Appendix C for examples of codes we dropped.

our central phenomenon. Bogdan and Bilken suggest considering the following types of codes [23]:

- Setting and context codes
- Participants' ways of thinking about people or objects
- Process codes
- Activity codes
- Strategy codes
- Perspectives held by participants

In our research we used all of these types of codes.

To initiate coding, one may look for phrases used repeatedly that indicate regularity in the setting. The start list of codes should be tied to research questions.

**Simple codes**

We tested several variations of coding schemes before we settled on the simple set that we will present in Section 4.3.2. Simple codes do not necessarily imply simple data, nor do they confer simple interpretation: simple codes permit the researcher to focus on a single phenomenon free of lengthy and possibly erroneous interpretation at the coding level. The more abstract the coding scheme, the more difficult it is to produce concrete statistics and to validate the work (e.g. to compute inter-rater reliability, which we describe below). Therefore, following our pilot study, we elected to use codes that were easiest to find in the data. Analysis was surprisingly revealing despite adhering to these simple conditions.

**To codify data**

The actual process of coding the data can be done with or without tools and in many different ways. In the initial stages, we viewed video while reading and marking up our transcription with xml style tags. This was fruitless because it was imprecise, inflexible to code changes (e.g. new or obsolete codes), and did not lend to deeper analysis. Moreover, our broad and inconsequential codes were often unrelated to each other and did not contribute to a more comprehensive theory. To work within a text-based transcript proved impractical. We needed an alternative.

We looked at several qualitative data analysis software tools, but they did not satisfy our needs. These were: an ability to review the video, a short learning curve, and an ability to adapt the process as new ideas emerged. We elected to code our data in Microsoft Excel on account of its rich feature set, intuitive interface, and macro programming. During the pilot study, we coded the data first on paper and input the codes in Excel. For the main study, we used Excel macros to improve the data coding process. In the latter stages of the main study, we exported our data in xml to a java-based qualitative data analysis tool (Qanal) for video review. We will discuss Qanal in greater detail in Section 4.3.4. Figure 4.2 illustrates a sample of our data coding in Excel.

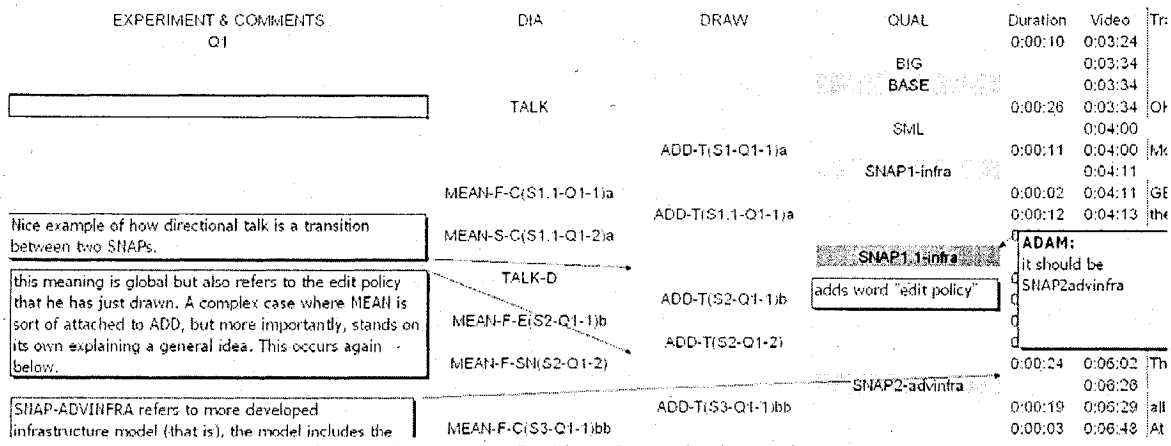

Figure 4.2: Data Coding Sample (Excel)

We elected to use a coding model of discrete, contiguous events to account for the entire session. We tracked the following information: the time and duration of the coded event, the code category (colour-coded and divided into columns by category of code), the corresponding dialogue, and the investigator's notes. Our data codification approach was sufficient to establish which codes were important and should be explored in more depth, how the basic phenomenon was manifested in the data, and how to capture and analyse data. We explore these findings in more depth in the Section 4.3.2.

### Category analysis

Codes are almost always related to one another, since they derive from a central phenomenon. As mentioned earlier, we selected to consider as our central phenomenon, or core category, a concept we called the 'snapshot.' We chose this because there was

sufficient evidence for it in the data, it was neither non-trivial nor overly complex, our industrial partner saw value from further investigation of it, and we surmised deeper investigation would yield strong, usable results. We performed a category analysis[19] by doing the following [33]:

- We assessed common threads and differences among the data associated with the snapshot codes. Repeated sequences provide good leads.
- We explored causal conditions. For example, as suggested by the grounded theory approach, we searched the protocol for places the participant had used the words "because" or "since".
- We wrote specifications of the strategies that resulted from the central phenomenon.
- We identified the context and intervening conditions.

During this process, we recoded data as required; doing so is a normal part of the category-saturation process. In our research, we made several changes to our coding during this phase. The results of our category analysis are described in the next chapter. We continued to revise our coding scheme until we reached the point of *saturation*, which we describe next.

**Saturation**

The qualitative researcher faces a significant challenge: how to know when coding is complete. Strauss [131] indicates that coding and recording are complete when all incidents can be readily classified, categories are saturated, and a sufficient number of regularities emerge. A researcher must balance deep analysis against the limitations of time, budget, and scientific relevance. In our research, we wanted to achieve comprehensive coding of all data and to reach a point where subsequent data analyses yielded no new categories. The following is an account of how we balanced data saturation and sufficient depth while following the constant comparative approach.

We began by coding two participants. We continued to code and recode these two participants at a rate of roughly 20:1 coding time to video time until no new codes emerged and the central phenomenon was well understood for these participants. Of course, two

[19] For more detail, please refer to Section 5.2.

participants do not provide a sufficient level of variety and to generalize from two cases is difficult. We were, however, able to find some emerging trends, and to develop an idea of what sort of extra analysis might yield further insight. To facilitate this analysis, we built prototypes of various tables and figures that displayed interesting information about the data. We term each of these 'data displays,' and discuss them further in Section 4.3.4.

We proceeded to code the remaining ten participants one at a time. New codes were rare, and trends began to solidify. As per the tenets of grounded theory, we focused only on coding during this phase, but we were mindful of where analysis might lead us. We had many coders[20] pass over data and verify coded data.

Figure 4.3 illustrates a sample of the change history for our day-to-day data. This listing of differences made it easy to monitor frequency and location of changes. To build the displays, we kept a historical record of all changes to our data; more specifically, on a day-to-day basis, we stored a copy of our data in a repository.

To examine the differences between these files and produce displays was a significant technical challenge. Displays forced us to improve our data integrity since much of our early data (prior to automated displays) was difficult to analyse programmatically. This problem was compounded by the fact we wanted to leave our data in its original form – that is, we did not want to introduce or remove changes as part of this phase, since our displays would not be representative of the data. Further compounding the problem was the evolution of the coding scheme. Though we originally intended the data change history displays would reveal the evolution, we did not anticipate the major programmatic work involved.

We settled on the core categories, ADD, REM, MEAN, TALK, and SNAP, and tracked changes in the form of additions, removals, moves and totals for codes within each category. We developed criteria to determine if a code was added, removed, or moved, and we required many trials to see accurate results. When we manually built the accurate results for comparison, we were close to dumping the programmatic solution, but pressed on because we wanted to share the process automation with future researchers.

[20] See Appendix A for Coder Profiles.

At a certain point, a data pass did not yield new codes or changes to current codes. Therefore, we elected to move from analysis to interpretation. We describe the results of the interpretation in the next chapter.

| 26/07/2005 | | | | |
|---|---|---|---|---|
| # ADDs | 69 | | | |
| # REMs | 1 | | | |
| # Moved | 5 | | | |
| # Label Changes | 0 | | | |
| Total | 75 | | | |
| Date | Type of Change | Code | Duration | Video |
| 26/07/2005 | MOVE | MEAN (S-C(S1-Q1-2)a -> S-W(S1-Q1-1)) (34 -> 29) | 0:00:00 | 0:03:55 |
| 26/07/2005 | MOVE | ADD (ADD -> B(S1-Q1-1)a) (34 -> 32) | 0:00:00 | 0:04:00 |
| 26/07/2005 | REM | TALK | 0:00:00 | 0:04:16 |
| 26/07/2005 | ADD | MEAN-S-C(S1-Q1-2)a | 0:00:00 | 0:04:20 |
| 26/07/2005 | ADD | ADD-B(S1-Q1-2)b | 0:00:00 | 0:04:22 |

Figure 4.3: Listing of individual differences between data files

It is very difficult to correlate coding time with quality of results, although if we performed many studies this correlation might be more obvious. The following rough phases reveal our progress with the data: Roughly the first 20% of the time was spent in 'initiation', whereby we began to understand the data, but had very little to speak of in terms of results. The next 15% was 'code breakthrough,' where we had an explosive increase in the number of codes. The next 50% was manual coding, in which there was modest code growth, a lot of manual labour in coding the data, ample discussion, as well as review of codes with multiple researchers. The final 15% was 'saturation,' in which new codes are rare, codes were refined, reviews became more technical, and data displays to test initial hypotheses were built.

**Inter-rater Reliability**

One may use inter-rater reliability to justify that codes reflect actual subject verbalizations. With this reliability mechanism, two coders independently analyse the coded data. When coding is complete, researchers calculate a coefficient of agreement; generally referred to as the kappa coefficient or Cohen's K. Researchers use Cohen's K to measure the proportion of agreement between two researchers who code the same chunk of data. Disagreements between coders indicate that code definitions need to be extended or

improved. When researchers disagree, they can review their discrepancies to improve the reliability measure.

Inter-rater reliability is necessary to validate that codes accurately fit the data and that coders do not introduce bias. The length of time to perform an inter-rater reliability check depends on what level of agreement is deemed suitable. We performed inter-rater reliability checks *comprehensively* across our coded data. For much of our data, we performed many checks. We isolated and performed inter-rater reliability for each individual code. Then we performed inter-rater reliability for the relationships between our ADD and MEAN codes. And finally, we checked the agreement regarding code times.

Inter-rater reliability can be computed as the percentage of codes that are the same from one rater to another. You would not expect 100% reliability; Seaman attained 62% agreement [118, pp.51]. Values lower than 60% indicate either the categorization needs revision or clarification, or else one or more of the raters need to re-visit their analyses.

Let A = number of agreements, and let D = number of disagreements:

$K = A / (A + D)$

$Reliability = K \times 100$

Let's walk through a real reliability scenario. For the purpose of simplicity, and because the coding of snapshots was the most critical coding activity, we will only look at snapshot-code reliability; though in typical scenarios we also checked our other codes, the relationships between codes, and code times. To make the process manageable we checked each separately, which led to improved integrity but also to lengthy analysis. We always performed inter-rater reliability checks with new coders, so in the following scenario one coder is an expert and the other is new to the project.

Two coders code the same piece of data. Then we run an Excel macro to generate data change history listings between the two, such as the one shown in Figure 4.3. This allows us to highlight the differences between the files. It is important to set criteria for changes between files, when, for example each coder might set the time for a code differently. Does this constitute disagreement and how much of a difference in time is reasonable? We only want to reveal disagreements that matter.

The point of the exercise is to determine what disagreements the two coders had, to determine if the disagreements are nomenclatural (inappropriate naming, e.g. tagging ADD-T instead of ADD-TL), conceptual (e.g. tagging a snapshot where the other coder insists there is no snapshot) or primitive (e.g. tagging a snapshot because of inappropriate cues – a result of immature coder knowledge). We do not count trivial disagreement in our measures; in fact, the majority of the interesting debate centered on the timing and category of snapshots.

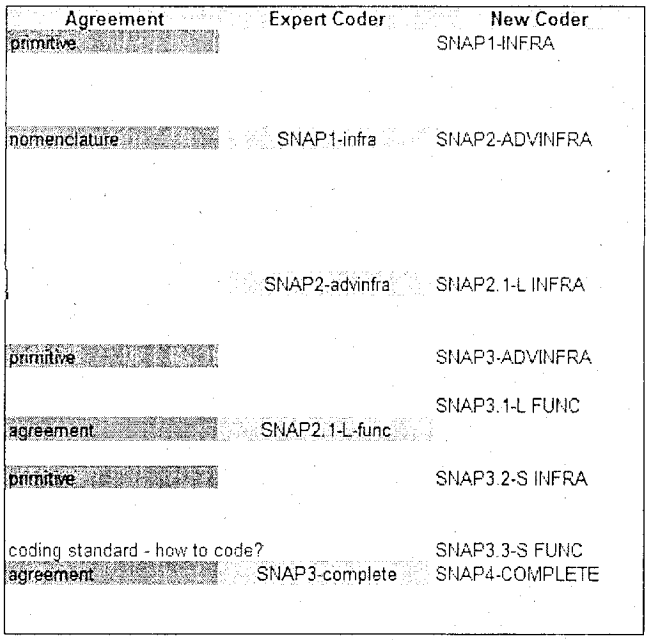


Figure 4.4: Inter-rater reliability sample between expert coder and new coder

Figure 4.4 illustrates the sample data for our scenario. A new coder is typically over-exuberant while tagging snapshots, as the right-column in Figure 4.4 shows. This is to be expected, the concept of a snapshot is not always easy to understand at first contact with the data. We handle various discrepancies as follows:

- Three cases of primitive snapshot codes are listed as 'primitive' in the left column. By reviewing the primitive codes, we reinforce the appropriate snapshot cues and the novice coder can avoid tagging primitive codes.
- Also in the left column, we see two cases of agreement and one instance of nomenclature.

- In one of the agreements, there is a misalignment because of timing, which is an irrelevant criterion.
- Aside from that there are two discrepancies, the expert tagged the code as SNAP2-advinfra, and the new coder tagged the same code as an infra, but lateral – which requires discussion. Why did the new coder think the discussion was lateral?
- The second discrepancy is the new coder's SNAP3.3-S FUNC, which was a code we missed in an earlier pass and provided us with a new perspective on the data. We had to agree on a new coding standard for similar events, and updated the coder's manual to reflect this change.

For the data in Figure 4.4, A = 3, D = 5, so Cohen's K = 3 / (3 + 5) = 0.375, or 37.5%. We found this to be typical for a new coder. After we excluded primitive differences, A = 3, D = 2, so K = 3 / (3 + 2) = 0.6. After further discussion to reconcile the two remaining disagreements, K rose to 1, or 100% agreement.

In practice, we cannot always reach 100% agreement: 'moral' debates ensue and occasionally, after hours of discussion, there is still no resolution and work must continue. In 85% of our reliability checks, K was initially in the range of roughly 0.50-0.70 and we improved it to the range of approximately 0.85-1. The mean increase in K following researcher discussion was 0.3. The remaining 15% of reliability checks account for new-coder situations in which training was the objective.

**Patterns & Theories**

We analysed our qualitative data to discover patterns, trends and generalizations. We sought evidence towards generalization as opposed to the individual differences between participants. As discussed earlier, the intended outcome of our analytic process is to generate a theory regarding a central phenomenon. The description, which includes rendering the study setting, participant perspective and specific evidence, is only one aspect of the theory. The theory is grounded in the views of the participants, and requires further grounding in the literature. The theory and the literature should contribute and contrast with one another harmoniously.

To convey qualitative theory through a narrative is popular, although displays, visual models, figures and interesting statistics can also be presented. In Section 4.3.4, we provide a detailed summary of the displays we used in our qualitative inquiry.

We developed the theory we call Snapshot Theory in response to the kinds of information added and removed, and the temporal patterns involved. As discussed earlier, our intent is that such a theory might aid tool developers in thinking about design. Tools currently tend to be designed with the final diagram in mind; if the tools were designed with the evolving state of a representation (including deletions) in mind, it may be possible to improve their user interfaces.

Any theory developed using this approach must be expanded and deepened by additional studies (preferably using the same or a related coding scheme). Other research needs to address the value of theory in terms of its application in practice. In Chapter 5, we describe our results.

### 4.3.1 Memos and Comments

The observation of collected data produced many insights into the structure and type of information contained in data. Further insights commonly arose as to how to approach the data and how to analyze its informational load. Marginal or appended comments began to connect different codes with larger conceptual units. While coding and recoding data, we used numerous memos and comments to track our rationale or doubts regarding the significance of codes, and sometimes we converted the memos into new codes. In this manner, we devised our initial coding scheme.

Following recommended practice, we ensured that comments were kept close to relevant transcription and codes. We also kept marked comments with the date, coder name, and contextual information such as the stage and relevance of analysis.

### 4.3.2 Coding Scheme

According to Miles and Huberman [93, pp.62], coding "is a way of forcing you to understand what is still unclear, by putting names on incidents and events, trying to cluster them, communicating with others around some commonly held ideas, and trying out

enveloping concepts against another wave of observations and conversations." In short, codes are tags used to assign meaning to data.

Because we had multiple coders, we found it necessary to have an unambiguous coding scheme. Codes became a common shared language between our coders, and we often had to refine the definition so coders could determine if data fit the code. Discussions between coders promoted the evolution of the overall conceptualization of the coding scheme and the evolution of the coding scheme was non-trivial. With the introduction or change to a code, we revisited the entire coded protocol to investigate the effects. We consequently found that coding changes during the interpretation or theory-building phases were extremely time-consuming and costly.

In undertaking the main study, we first coded our protocol according to a series of codes from the pilot study (Section 3.5): ADD, MEAN, SNAP, TALK, etc. We then established the snapshot as the core unit of analysis and developed the coding scheme in more depth around this core unit. The revised codes were more comprehensive and sensitive to subtleties in the data. Our codes were explicitly tied to our choice of media. The ADD, MEAN and SNAP codes are dependent on video. We will provide justification for this dependency in the coding scheme. In contrast, the TALK code is not dependent on video; we can find instances of this code in the protocol.

Four independent researchers agreed that the pilot coding scheme was an appropriate classification of the data *in general*. However, it was clear that the pilot codes were underdeveloped. We therefore made the following types of changes to our coding scheme:

- To aid the analysis of codes, we devised a unique identifier for each usage of a code. As a result, we could distinguish among groups of similar codes, which made memo writing and analysis easier.
- We explored the relationships between categories. For example, we examined whether ADD codes corresponded with particular MEAN codes and likewise whether ADD or MEAN codes corresponded with particular SNAP codes. We used a relationship identifier in order to analyse these types of relationships.
- We further developed the coding scheme to be objective, context independent, exhaustive (to cover the entire data set) and easy to record.

- In the pilot study, we only tracked whether drawing was occurring, but did not track what was drawn. In the main study, we used generic drawing shapes (i.e. box, line, etc.) as opposed to software terms (e.g. UML) in order to maintain context independence. The drawing codes in Section 4.3.2.1 describe the types of shapes and provide examples.
- The data revealed distinct *sub-categories* not present in the pilot coding scheme. For example, in the pilot, we broadly tagged snapshots, but in the main study, we became aware of different sub-categories of snapshot that exhibited different characteristics, e.g. weak or helper snapshots. All codes are written to identify both category and sub-category (e.g. MEAN-S-E has category MEAN and sub-category S-E). We tagged each ADD, REM, MEAN, TALK and SNAP code with sub-category. No codes are marked with only high-level category.
- The syntax for coding in the pilot study was to write the code in its abbreviated form, e.g. "SNAP" for snapshots. In the main study, the addition of unique identifiers, relationship identifiers and sub-categories led us to write the code in its abbreviated form with additional details. For example, an ADD code, as shown in Figure 4.2, might take the form "ADD-T(S1-Q1-1)a", where "ADD" refers to the category, "T" refers to the sub-category (text), "S1-Q1-1" refers to the unique identifier (the first add of the first snapshot and the first question) and "a" refers to the relationship identifier (e.g. a MEAN code with relationship identifier "a" refers to a descriptive explanation for this drawing element).
- We extended our pilot codes to comprise deeper behavioural elements in the data.
- Certain codes became obsolete because they were:
    - Subsumed into new categories (e.g. codes from our pilot study were subsumed in our main study: MBED was accounted for within the new MEAN classification, SET within the new TALK classification, etc.);
    - Ineffective in the investigation of the snapshot as the core unit of analysis.

In the following section, we describe in detail the coding scheme illustrated in Figure 4.5. For each code, we provide explanation, rationale and an example, and discuss some of the events or incidents that made these codes apparent during analysis.

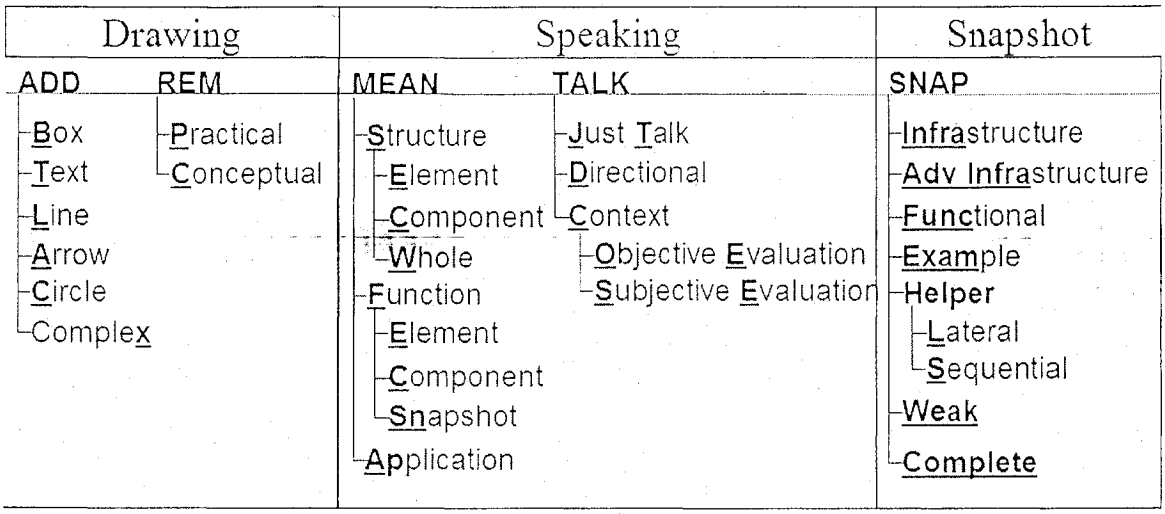


Figure 4.5: Code Classification

#### 4.3.2.1. Drawing Codes

**Add Detail** (ADD): The participant *draws new material on the whiteboard.* The investigator who reviews video with no sound can accurately code drawing instances for the entire data set (investigator sees shape, investigator marks data). Though a participant may engage in dialogue while they draw, if the pen is on the whiteboard, we apply the ADD code. We deal with situations where a participant is drawing for a very long time by tagging subsequent individual shapes. At first, we limited the ADD code to a single gesture (e.g. a box, an arrow, a line); hence, every distinct shape was tagged with its own code. However, our participants regularly used the same combination of gestures (e.g. an arrow drawn from a box), so we extended our coding to include these combinations. We use a relationship identifier in the coded data to mark if an ADD code corresponds to participant dialogue (i.e. a MEAN code). The ADD codes are simple and may be coded by investigators from different domains and levels of education.

The sub-categories of the ADD category used to indicate shape are as follows:

i) (B)ox (ADD-B): A *rectangular shape*, which typically contains a label that designates the concept the box represents. In our study, this shape had squared corners, though we would also have permitted rounded corners[21]; however, if the shape is elliptical, we use the 'ADD-C' code, below. Boxes typically represent such things as software layers,

[21] If we had seen triangles or parallelograms, we would have revised our coding scheme, but the box shape was common and we saw little variance.

components, objects, classes, and user interface elements. For example, a participant drew the boxes in Figure 4.6 to indicate software layers.

'Right at the base there's Eclipse. And then, layered on top of Eclipse…'

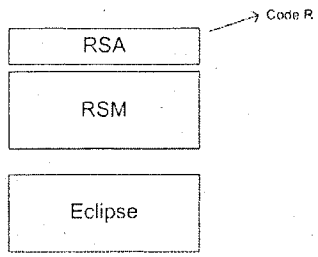


Figure 4.6: Drawing Example (box)

ii) (T)ext (ADD T): A *label* added to the diagram. For example, a participant described the label in Figure 4.7 as he added text to the diagram.

'There's a transform provider…'

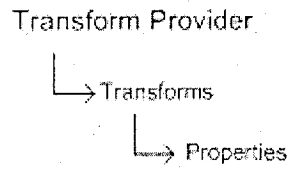


Figure 4.7: Drawing Example (text)

iii) (L)ine (ADD L): A *non-directional link* between diagrammatic elements. The example participant used a line in Figure 4.8 to indicate the connection between the controller component and view component in a model-view-controller architecture.

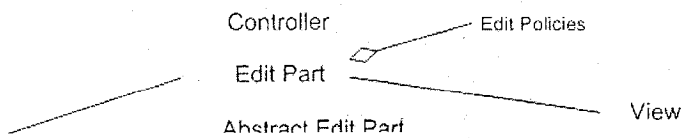


Figure 4.8: Drawing Example (line)

iv) (A)rrow (ADD A): A *directional link* between diagrammatic elements which indicates function or inheritance. The example participant used the arrow in Figure 4.9 to

indicate that the AbstractGraphicalEditPart is a subclass of the AbstractEditPart. Later, the participant continued the inheritance hierarchy with a lower-level class, GraphicalEditPart.

'From this comes the abstract graphical edit policy, which brings in a concept of figures…'

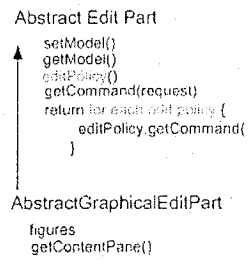


Figure 4.9: Drawing Example (arrow)

v) (C)ircle (ADD C): An *elliptical shape*, which may contain text, and represents a concept in the diagram. For example, a participant used circles to express how products overlap in Figure 4.10.

'If they were diagrams it would be ReqPro here, and RSA here…'

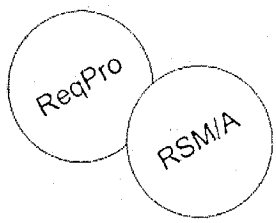


Figure 4.10: Drawing Example (circle)

**Remove Detail** (REM): The participant *erases* some or all of whiteboard material (i.e., additions or whole snapshots). The participant may engage in dialogue while removing material.

The REM code breaks down into two subcategories, which address the rationale for removal, *practical* and *conceptual.*

i) Practical (REM-P): The practical code marks a case when participant removes whiteboard material because of *lack of space* or *poor organization.* For example, the

participant displaced a component from the main model in order to present the component on a bigger part of a whiteboard.

ii) Conceptual (REM-C): The conceptual code marks a case where participant removes whiteboard material because of *conceptual reasons*, such as *complexity of the abstraction* or *disharmony between knowledge and explanation*. For example, one participant realised his explanation and drawing were further advanced than his audience's understanding, stated "sorry, before that...", erased a recent addition in order to backtrack, and drew something else. He realised that he had moved too quickly and removed detail in order to first provide other needed concepts.

#### 4.3.2.2. Speaking Codes

We divide moments when the participant speaks into two categories: *talking*, and *meaning*.

**Talking** (TALK): The TALK code is a general code that marks instances when the participant references contextual aspects of the explanation; for example, the participant expresses a *general opinion* with respect to the software system, usual approaches to the system, the system's structure or functionality, *how experienced and knowledgeable* he or she is with regards to the problem space, or how he or she *feels*. The participant also provides *directional focus for material*, as well as *bantering and anecdotal expressions* that have less to do with the substance of the explanation. From the operational definition perspective, TALK includes the participant speaking, while he or she is not drawing or discussing prior or future diagrams.

The TALK code breaks down into three main sub-categories, *just talk*, *context*, and *directional*. TALK sub-categories are mutually exclusive.

i) Just talk (TALK-JT): The "just talk" code represents the participant's *general statements* (e.g., bantering, anecdotes, etc.) that improve explanation flow, but give *no substantial contribution* to the comprehension of the issue. Consider the following anecdotal expression:

'Now this is not reverse engineering so there is a difference there. In reverse engineering you're taking your Java model and creating a UML model out of it.'

ii) Context (TALK-C): The context code represents the participant's perception of *contextual aspects of the domain*, either the *state of the system or personal states*. We divide the context code into two sub-codes, *objective evaluation* and *self-evaluation*.

ii-1) Objective evaluation (TALK-C-OE): Statements that provide *general estimates, opinions or biases* about issues such as state of the system, system functionality, structure, or other practical aspects. Consider the following broad assessment, free of specific details:

'It tries to tie in a lot of the components that Eclipse gives us because they are really good, they give us a lot of bang for their buck. Not perfect, but pretty good.'

ii-2) Self-evaluation (TALK-C-SE): Statements that reveal an *evaluation of the participant's experience, knowledge, or personal state*, such as lack of integrated knowledge, time constraint, lack of preparedness, etc. Consider the following self-evaluation of the participant's knowledge with respect to a particular topic:

'OK. I'm going to be a little bit fuzzy around this, because I don't know, exactly.'

iii) Directional talk (TALK-D): The directional code represents the participant's *direction of the recall process* and/or *direction of the audience's attention* toward the wholeness of a structure or toward the following topic in the explanation *(e.g. 'that's the general structure' or 'let's take a step back')*. Directional talk underlines *conceptual wholeness*, or *announces upcoming content*; for example, *'Now, profiles.'* indicating upcoming content in the explanation.

**Meaning (MEAN)**: The MEAN code marks instances of description or explanation of the structure, function, or application of the software system the participant has explained, will explain, or is currently explaining, at different levels of abstraction that include model elements, model components, or the whole system. As in Section 4.3.2.1, we use a relationship identifier in the coded data to mark if a MEAN code corresponds to whiteboard drawing (i.e. an ADD code). The MEAN code breaks down into three main sub-categories: structure, function and application. The structure and function categories are then divided into levels of abstraction. MEAN codes are mutually exclusive. Before we describe the MEAN sub-codes, we will describe some of the analytical challenges we faced in developing this code.

Qualitative researchers interpret data. Sometimes, the interpretation is subtle and unintentional, so a primary goal in qualitative research is to clarify bias and to be critical of how the data is interpreted. We must, therefore, explain an important detail with respect to our coding of meaning. We can code the ADD codes through the observation of silent video – but, only with great difficulty can we code instances of the MEAN code through the observation of sound alone (e.g. listening to audio tapes). When, in the course of our research, we coded MEAN instances, we tried to find support in the video and the audio. We were interested in 'the software system the participant has explained, will explain, or is currently explaining' and we had to build evidence through both visual cues (e.g. participant motions to whiteboard space he or she previously used) and audio cues. Furthermore, we relied on video to indicate the level of abstraction where this was not evident in the audio protocol. For example, the participant would often visually indicate the level of abstraction by pointing to a single element, motioning to a combination of elements, or waving at the entire whiteboard. Therefore, we were dependent on the audio and video data when we applied the following codes.

A further challenge was the subtle distinction in certain cases between structure and function. For example, the participant draws and explains a user interface; is this structure or is this function? Though the answer is dependent on the specifics of the situation, if the participant describes how a user interacts with the interface then the participant describes function, whereas if the user describes the user interface elements as a part of the system then the participant describes structure.

Another challenge arises from the varying levels of abstraction that might, from one drawing to another, be represented by the same drawing element. For example, in one drawing, a method is represented by a single element, but in another drawing, the method is represented by the whole model. Or a product such as Rational Software Architect (RSA), which in an architectural drawing is the whole model, might in a product-line drawing be a single element among other products. In the following sections, when we use the terms 'element,' 'component' and 'whole model,' we use them in the context of the level of abstraction within the diagram in which they are drawn – hence the dependence on video data.

i) Structure (MEAN-S): The structure code represents statements in respect to the *composition of the model.* We divide the structure code into three sub-codes: *structure element*, *structure component* and *structure whole model.*

i-1) Structure Element (MEAN-S-E): The structural element code represents statements in respect to the *lowest-level model constructs*. Consider the following reference to a single element of Figure 4.10:

'From what I do know, ReqPro is a requirements gathering tool…'

i-2) Structure Component (MEAN-S-C): The structural component code represents statements in respect to *several elements of the model.* Consider the following explanation in which the participant relates three model elements:

'We have three common tabs, that we support and supply with a transform GUI...'

i-3) Structure Whole Model (MEAN-S-W): The structural whole model code represents statements in respect to the *model structure described as a whole.* Consider the following summation of an entire model:

'So this is the big picture of all of our tools right now,'

ii) Function (MEAN-F): The function code represents statements in respect to the description of *model processes, mechanisms, or inner-workings.* We divide the function code into three sub-codes, *function elements*, *function component*, and *function snapshot* (i.e. general functionality).

ii-1) Function Element (MEAN-F-E): The function element code represents statements in respect to the description of the way in which a *single element functions*. Consider the following statement, which explains what a single graphical element, the transform provider, can do:

'A configuration instance: a configuration instance is a record that contains the information in a parameter…'

ii-2) Function Component (MEAN-F-C): The function component code represents statements in respect to the description of the way in which a series of elements co-

function. Consider the following statement which explains how a set of extractors work within RSA:

'...various extractors which take information out and then feed them into rules that actually do the mechanics of performing the transformations of various types.'

ii-3) Function Snapshot (MEAN-F-SN): The function snapshot code represents statements in respect to the function of the entire model at the participant's intended level of generality at a given moment in the explanation. The level may vary from enriched basics of the highest level of abstraction (i.e. high-level infrastructure) to a model developed far enough to generate cognitive insight. The model may not yet be complete, but the participant describes the meaning of the model function in terms of its constituent components and wholeness. Consider the following statement, which explains the general mechanics that provoke insight with respect to the model function at a given level of generality:

'So any time you make any change, the user gets immediate feedback. If he's broken something, it will go from blank to having an error; if he's fixing something it will go from an error to blank, or removing it – there can be more than one error in this particular set.'

iii) Application (MEAN-A-P[22]): The application code represents statements in respect to the practical value or real-world implications of a model or the usage for a model. Consider the following example of how a user can engage the system to satisfy practical purposes:

'And when you say, I want to run a transformation, and I want the output to go to this here, model, file, project folder, or something like that.'

#### 4.3.2.3. Snapshot Codes

**Snapshot** (SNAP): A snapshot marks the culmination of an interval of time that contains enough information about a model to reveal meaning. The snapshot is defined by its conceptual wholeness at a given level of generality. Often a researcher may identify a snapshot based on the fact that a participant pauses to reaffirm something, e.g. a particular structure or a set of mechanics within a system's representation. A snapshot is usually

[22] Please note, the code should be MEAN-AP; we kept MEAN-A-P for legacy purposes, but they are synonyms.

revealed after a verbal or diagrammatic summation of a model. Our normal practice in this thesis will be to provide code samples or vignettes; in the case of snapshots, however, we will provide in the analysis and interpretation sections both a rationale for their use and support for their existence.

The snapshot category has sub-categories we divide by category (*infrastructure*, *advanced infrastructure*, *functional*, *example*), by helper status (*lateral*, *sequential*) and by quality (*weak*, *complete*). As previously stated, all codes are marked with sub-categories, so we do not permit a SNAP code without category, helper or quality, though a SNAP code may have one, two or all three (e.g. a lateral complete infrastructure snapshot is possible.)

Snapshots are different from TALK and MEAN codes in that sub-categories are not mutually exclusive. The reason is as follows: as the notions of helper and quality emerged, we realized a snapshot can be both lateral and functional at the same time. We did not want to extend the number of codes (i.e. to include lateral functional, lateral example, etc.) as this would hinder analysis. Because of the lack of mutual exclusion we increased programmatic complexity and coding complexity.

A snapshot's *category* is consistent with the way the software is typically described in terms of structure, function and application, though we did not originally intend this. A *helper* snapshot stems from a prior snapshot and is a supplement or further elaboration by which a participant builds upon a preceding snapshot using independent or sequential sub-models. A helper snapshot is non-essential to an explanation, but is still conceptually whole. *Quality* refers to a snapshot's level of "wholeness" – whether it is sufficient for insight into an entire concept, or intended as sufficient but lacking specific details required for wholeness.

We tag snapshots with sequential identifiers (e.g. SNAP1, SNAP2, etc.). In the case of lateral snapshots, we use the original snapshot identifier followed by ".1", e.g. (SNAP2.1) to demonstrate that SNAP2.1 is a lateral discussion branching from SNAP2. Along the helper branch for SNAP2.1, we increment sequential snapshots as SNAP2.2, SNAP2,3, etc., and further lateral branches, SNAP2.3.1, SNAP2.4.1, etc.

**Snapshot categories**

i) Infrastructure snapshot (SNAP infra): The infrastructure snapshot represents a moment in which the participant reveals the model at its *highest level of generality*, typified by the *visual structure with only basic structural elements shown.* The infrastructure snapshot is the most abstract form of snapshot, stripped of details, specifics, processes, etc.

ii) Advanced infrastructure snapshot (SNAP advinfra): The advanced infrastructure snapshot represents an *infrastructural model* in which *at least one of the basic elements is structurally developed.*

iii) Functional snapshot (SNAP func): The functional snapshot represents a moment in which the participant conveys the *processes, mechanics, or inner-workings* of the model. The functional snapshot may capture the participant's explanation of a function extracted from the main model.

iv) Example snapshot (SNAP exam): The example snapshot represents the moment in which the participant expresses the *applied usage of the model.* The participant articulates *how the model is used in practice.*

**Helper snapshots**

i) Lateral helper snapshot (SNAP L): The lateral helper snapshot instigates a discussion branch to advance the meaning of a prior core snapshot. The lateral helper is non-essential to the core discussion, but helps build more complete insight into the explanation. The lateral helper involves the construction of an aside in the form of a new model.

ii) Sequential helper snapshot (SNAP S): The sequential helper snapshot is a special kind of lateral helper snapshot which continues the discussion branch instigated by the lateral helper or continues the branch from a previous sequential helper. A coder cannot have a sequential helper snapshot without first having a lateral helper to instigate the discussion branch.

**Snapshot quality**

i) Complete snapshot (SNAP complete): The complete snapshot represents the moment in time in which the participant reveals the *full meaning of the cognitive model.*

The *participant completes his or her depiction of software understanding*. A complete snapshot may follow one or more snapshots and may therefore summarize or close out an ongoing topic. Or, a complete snapshot may be independent of other snapshots; that is, a concept that is developed and fully described in the space of a single snapshot (we call this a "solo-complete snapshot"). Lateral branch discussion can also lead up to a complete snapshot, which also suggests that a complete snapshot can follow other nested complete snapshots.

ii) Weak snapshot (SNAP weak): The intention of the participant is to communicate meaning or understanding using a regular snapshot but uses instead the weak snapshot which *lacks structural and/or functional elements* necessary for complete comprehension of the model.

### 4.3.3 Tactics for analysing coded data

In order to move from codes to patterns, we employed a number of tactics for analysing coded data. These include: noting patterns, seeing plausibility, clustering, metaphors, counting, making contrasts and comparisons, partitioning codes, noting relationships between codes and noting outliers.

*Noting patterns* can be both simple and complex. A simple case is the observation of recurring sequences in the coding scheme. Because, as Figure 4.2 shows, our data is colour-coded, many sequences are readily apparent to the keen observer and can be easily checked against other cases. More complex patterns are found through matrix displays, which reveal trends or combined sequences of video data that require multiple passes. In the course of our research, noting patterns was the most critical tactic for generating meaning.

As Miles and Huberman [93, pp.246] identify, "the history of science is full of global, intuitive understandings that, after laborious verification, proved to be true. *Plausibility*, with intuition as the underlying basis, is not to be sneered at." On the other hand, the researcher who is too eager to finish analysing may seek refuge in plausibility. Section 3.4 sheds light on how the initial plausible notion of temporal details and snapshots emerged. Likewise, similar cases of plausibility were not uncommon for tagging data: The sequence

was, a) see behaviour, b) investigate against other cases to note recurrence, and c) tag occurrences with a code to investigate plausibility at deeper levels.

Plausibility also guided us in decision-making about how to structure and handle data (i.e. specifics of Excel coding, storage of Qanal history). Our analysis evolved, but there was always a starting point when we said, for example, "is it plausible to record the dependence between drawing and speaking codes." We logged decision rules for data entry and handling to preserve unity among coders.

We used *clustering* to inductively form new categories and iteratively sort events from different participants into those categories. For example, when examining a particular pattern, we would group all instances of that pattern in Qanal for subsequent review.

With *metaphors*, we need to be cautious about overloaded meaning. A primary example of the use of metaphors is the language for our codes. The term 'snapshot' is a metaphor. Another example is the term "lateral helper". The term 'lateral' suggests that the content of a snapshot is non-essential, and this is perhaps literal enough. However, the term 'helper' suggests that the content of the snapshot is helpful, and this marks an abstraction and highlights an assumption. The assumption is that this content supplements other content and is useful. Consider the ambiguity of calling one snapshot a helper, but not others. Does the non-lateral or 'main line' of snapshots not 'help' if these snapshots contribute to audience understanding? We needed to consider details as such when making the decision to use the term 'helper' to denote snapshots that help the meaning of the main line.

In short, we used metaphors to connect our theory of software explanation to our findings that related to temporal details. From informal discussions with other software researchers and practitioners we found that our use of metaphors, in particular the snapshot metaphor, ignites the imagination as to the impact of temporal details theory. The participants in our studies also regularly used metaphors to share their meaning. Such metaphors helped us define the boundaries of discussion and therefore helped us to identify snapshots.

*Counting* is used frequently in qualitative data studies of software. In our research, we produced a significant volume of quantitative code counts, which allowed us to hone in on

the important or significant themes in the data on which to base our theoretical findings. This may seem counter-intuitive to the essence of qualitative research, yet counting quickly revealed trends in a large set of codified data and allowed us to explore our hypotheses about the data. That is, when we identified recurrence, we could observe if certain behaviour recurred in a specific way.

A common practice in qualitative research is to group instances or sequences of data and *make contrasts and comparisons* to determine meaning. In our research, we compared instances of codes, specifically instances of snapshots. We also contrasted participant data using tables and charts and compared instances of day-to-day coding to determine if saturation had occurred. We did not contrast the sample from our pilot study with the sample from our main study on account of time constraints.

Though the course of qualitative analysis moves toward integration, there are times when *differentiation* is critical. The transition between the initial pilot study and the main study is an example of such differentiation. We partitioned several codes into many sub codes, which freed us from what Miles and Huberman call "monolithic simplicity" [93, pp.254] and provided us with deeper focus. Likewise, as analysis proceeded, we found the need to partition yet further – the lateral and sequential snapshots are examples of the partitioning of the helper snapshot code. We differentiated enough to allow ourselves to distinguish between codes and thus relate the theory, but not so much to overwhelm the analytic process with complexity. In contrast, we rarely subsumed particulars into the general, that is, we rarely found codes that belonged to a more general class.

Once the data were coded, it was common to *note relationships between codes*. In particular, we noted the correlation between ADD and MEAN codes, between ADD and SNAP codes, and between MEAN and SNAP codes. Matrix displays, as will show, are an efficient way to demonstrate relationships.

We often found and challenged *outliers* in our data – chunks of data that did not correspond to our coding scheme or general expectations. Sometimes we found the outlier through odd participant behaviour (e.g. not drawing for long periods of time). Other times, we found snapshots that failed to invoke the kind of insight we expected because they lacked structural or functional foundation (these evolved into what we call 'weak

snapshots'). Other outliers were strange drawings that did not fall into our scheme. We made slight modifications to accommodate these outliers, but this was probably not necessary because it did not affect the findings. In general, the outliers were very easy to locate. We generated data displays that clearly illustrated a) problems with the coding scheme; b) problems with our automatic data-display generator source code; and c) problems with the video that led to strange time sequencing and therefore strange displays (which we had to correct). If a chart or table looked dramatically different from neighbouring or prior displays then we took interest and either argued over the place of the outlier in the overall theory or corrected some conceptual or technical issue that drew our attention to the outlier.

### 4.3.4 Instrumentation

The researcher is the primary measurement device in qualitative research. That said, when working with codified video data, additional tools help to make tasks repeatable and easy. For example, when we clustered snapshots of a particular category for review, we used the coded data to tell us which chunks of video we needed to look at and performed manual seeks to the appropriate time sequence. The problem is that when analysing video, this operation must be performed very frequently – and at the outset of our research there was no effective way to a) expedite the operation, b) capture the operation, or c) replay the operation.

To deal with this problem, we considered either using tools designed for qualitative analysis of video data or building our own tool. Existing tools had a number of problems. Firstly, they did not support the import of data we had already coded in Excel. Secondly, the commercial tools are either rich with irrelevant features (bloated), not extensible to our needs (poor customizability), or require extensive training.

Our decision, therefore, was to build our own tool in Java – we call it Qanal, short for "Qualitative Analysis Tool" – to support the storage and review of codified video data. Qanal, in conjunction with Excel macros, provided immeasurable benefit to our research. Our tool requires training too; however, because Qanal is tightly designed for our requirements, the difference in training time between Qanal and existing tools is significant.

The features of this tool are as follows. It can:

- Export chunks of coded data from Excel into an XML data file (an Excel macro performs the export to XML)
- Import XML data file of codes as an 'exploration' within Qanal
- Explore operations, i.e. VCR playback of data files
- Collect explorations as local history within a hierarchy
- Access local history in order to load explorations and replay them. This is useful to the analyst who:
    - Confirms a proposition against prior explorations;
    - Needs a refresher of how events took place months after exploring data;
    - Wants to share findings as a story with others.

An XML file has multiple codes. Each code within the XML file has the following fields:

<Video>: Video file to which the code corresponds

<Code>: Code identifier

<Time>: Code event start time

<Duration>: Code event duration

<Protocol>: Corresponding dialogue

<Comment>: Investigator notes

Figure 4.11 is a screenshot of Qanal. We will presently describe the user interface of the tool.

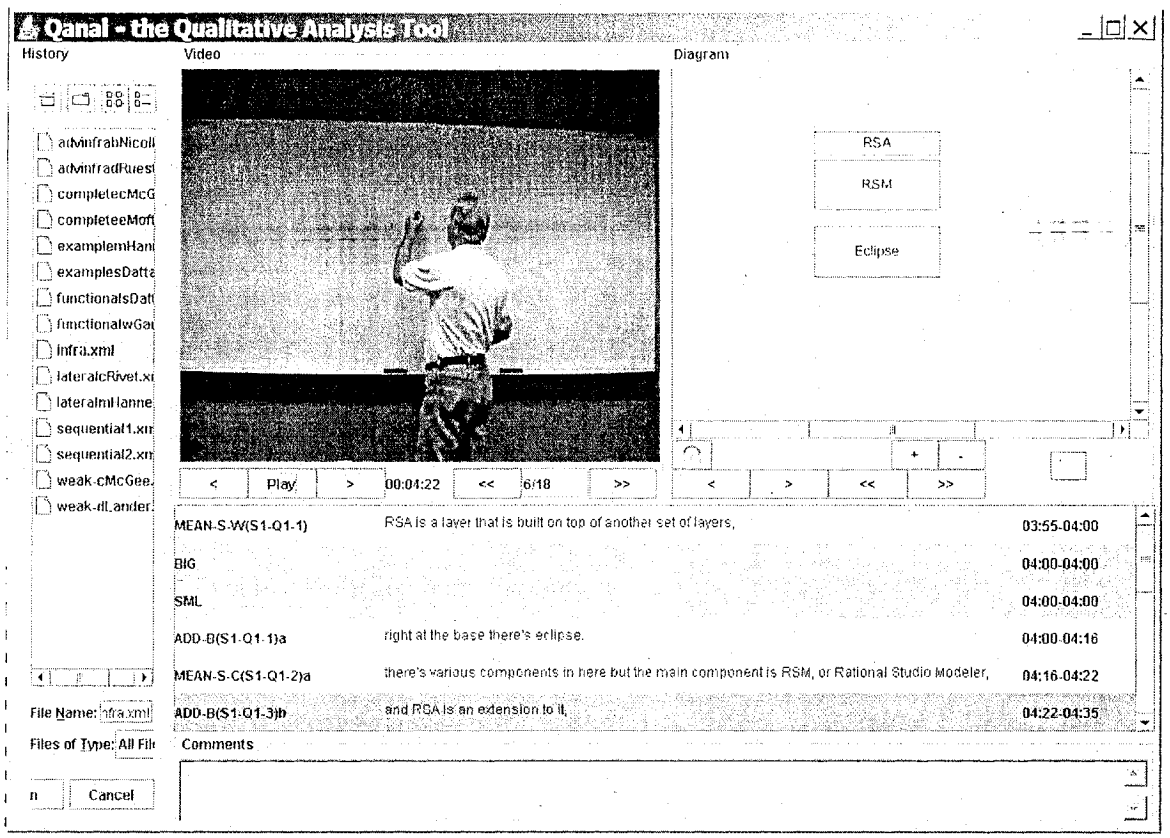


Figure 4.11: Screenshot Qanal: the Qualitative Analysis Tool

### History Pane

The history pane is a file hierarchy system that mirrors the local drive. When Qanal is started, the history pane opens in the Qanal history folder. The Excel macro for exporting to XML saves the XML file in the Qanal history folder. From the history pane, a user can load an exploration to play in Qanal, move explorations to other file locations, create new folders, and rename exploration files. The reason we always show the history pane is because accessing various explorations is a regular analysis operation; an auto-hide feature is a reasonable enhancement.

### Video Pane

The video pane plays and explores video clips. When an exploration is loaded, it does not start to play automatically. The user clicks the Play/Pause toggle to play or pause the video. Each click of the < and > buttons which surround the Play button moves the video location back or ahead in ten second increments. The time, e.g. 00:04:22 from Figure 4.11, is the actual video time (four minutes, twenty-two seconds), not the time since the

exploration was started. Each click of the << and >> buttons moves the video location back or ahead by code. The code indicator, e.g. 6/18 in Figure 4.11, indicates that we are presently viewing the sixth code of eighteen codes. The video pane uses the Java Media Framework (JMF) and runs video at a smooth pace. Resizing the window maintains the aspect ratio of the video, and continues to smoothly run video.

**Diagram Pane**

The diagram pane plays and explores diagram files. We composed these diagrams files in Microsoft Visio to match the participant's whiteboard diagrams. We saved the diagrams in folders for each participant in .gif format with file names that corresponded to the code identifier. If the video is ever unclear, or we simply want to follow the diagrams instead of video, the diagram pane will continue to update as video plays (the diagram pane shows the diagram file that corresponds to the code under observation). The lock/unlock toggle allows the user to lock the diagram to the video (in which case when the code changes, the diagram changes), or unlock it (in which case, the user can freely navigate forward or backward through diagrams and the diagram will not change when the code changes). The scroll panes allow the user to view diagrams that do not fit into the diagram pane. If the diagram is larger than the pane, the thumbnail image allows users to see which part of the diagram they are currently viewing in the diagram pane. The user can also move the red rectangle within the thumbnail image, which moves the viewable region in the diagram pane. The + and – buttons zoom out and zoom in to increase or decrease the viewable region in the diagram pane. The < and > buttons change the diagram to the last or next diagram and the << and >> buttons change the diagram to the first or last diagrams. The diagram pane is useful to remind the user how the diagram is about to unfold and to zoom in to the intimate details of the diagram when they are unclear from the video.

**Code Pane**

The code pane illustrates the codes for an exploration. Each code displays the code identifier, the protocol, and the start and end video time for the code. Each code is colour-coded to match the colour-coding in Excel (colour-coding is simple to code: parse the code prefix from the code identifier and compare against a properties file). Qanal generically handles and accepts all codes. Single-clicking on a code moves the video to the appropriate

clip and updates the video pane (i.e. the video, the time and the code indicator). Each code has a uniform row height to make visual parsing of information easy. However, the downside of this is that there are times when the protocol expands beyond the bounds of the code. To address this, we implemented tool tips which reveal the complete protocol when the user moves the mouse over a code. The scroll bar allows the user to navigate through the complete set of codes for an exploration.

**Comments Pane**

If the researcher has embedded comments for a particular code, they appear in the comments pane while the code is playing.

**Under the Hood**

Qanal is written in Java, and uses the Swing toolkit for its user interface, the Java Media Framework (JMF) along with native codecs to play back video, and the Java Simple Access to XML (SAX) Parser to parse XML exported from Microsoft Excel. We implemented Qanal with a Model-View-Controller (MVC) pattern, which brings two advantages: maintainability (many software developers are familiar with MVC) and flexible improvement to user interface design.

The JMF provides the video-playback capabilities. JMF is integrated with the Swing toolkit on the front-end, and uses native, open-source codes on the back-end to boost performance. This gives Qanal an extensible selection of video formats; researchers who wish to use an unsupported video type may simply download and install a new codec, and Qanal will be capable of playing that video type.

### 4.3.5 Pattern Coding and Displays

In this section, we describe the displays we built as part of the analysis phase. In the next chapter, we describe our interpretation and results that derive from the displays. For each display, we offer a data sample, give our rationale for how we filled the display, and describe the tactics for generating meaning from Section 4.3.3 to give the reader a taste of how our qualitative data analysis occurred. We illustrate tactics in *italics* in the forthcoming text. Our inquiry proceeded along the following path:

- time-ordered sequencing of codes;

- investigation of codes;
- investigation of code relationships;
- investigation of snapshot relationships with codes;
- investigation of snapshot relationships with other snapshots.

In addition, we will briefly describe the other displays we built to assist the analysis process.

**Time-Ordered Matrix**

We applied the time-ordered matrix (c.f. Figure 4.2) to codify our data. As qualitative researchers, "we are always interested in events: what they are, when they happened, and what their connections to other events are (or were) – in order to preserve chronology and illuminate the processes occurring." [93] This type of display provides event time, duration and sequence, and we can look for patterns in sequences of events.

In the data sample for the time-ordered matrix illustrated in Figure 4.2, the coded data is divided into seven columns. The first column contains experiment comments (e.g. "Q1" indicates that the investigator asked question one). The second column contains dialogue codes. The third column contains drawing codes. The fourth column contains qualitative codes, including snapshot codes. The fifth column contains event duration. The sixth column contains event time, recorded to the second. The seventh column contains transcription, tied precisely to the event times. Event codes are coloured by type to ease reading and pattern identification.

Our time-ordered matrix yields at least two benefits. Firstly, to read and to manipulate codes was easy in this format. Given the lengthy coding process, this first benefit was realized throughout analysis. Secondly, by entering the time-ordered sequence in Excel we were able to programmatically generate all subsequent displays – we could change codes and regenerate displays to obtain an accurate depiction of the current data set. This second benefit was invaluable, because we found that as we built and interpreted displays, we were required to make many revisions to the codes. In fact, to build a display programmatically we created an integrity checker that indicated human coding flaws. It was quite natural to build a display by hand the first time to allow the investigator to understand the criteria and

decision rules for each display. However, future versions of a display could all be based on current data; to build them by hand would have been labour-intensive and error-prone.

**Drawing-Speaking Independent Matrices**

To examine the drawing and speaking codes we built matrices that illustrate the frequency and duration in terms of code counts and the proportion of each code across all participants. This type of display may be considered one of the foundations of our analysis. For example, in order to *note relationships between codes* we need to first understand the context within a study in which a code exists. These tables illustrate the dominance of particular codes within our sample, and therefore lead us to form hypotheses for further investigation.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD-B(* | 5 | 13 | 40 | 5 | 24 | 17 | 14 | 33 | 20 | 23 | 13 | 12 | 219 |
| ADD-L(* | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 3 | 3 | 0 | 5 | 1 | 15 |
| ADD-T(* | 15 | 28 | 1 | 1 | 3 | 16 | 39 | 16 | 14 | 7 | 28 | 42 | 210 |
| ADD-A(* | 3 | 2 | 0 | 3 | 0 | 7 | 8 | 10 | 2 | 3 | 6 | 0 | 44 |
| ADD-BA(* | 2 | 0 | 1 | 3 | 0 | 0 | 1 | 0 | 3 | 0 | 0 | 0 | 10 |
| ADD-TB(* | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| ADD-LB(* | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 5 |
| ADD-AT(* | 4 | 9 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 2 | 27 |
| ADD-C(* | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 6 |
| ADD-TL(* | 2 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| ADD-LA(* | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| ADD-TLA(* | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **SUM** | **38** | **59** | **42** | **19** | **27** | **44** | **62** | **63** | **43** | **39** | **52** | **60** | 548 |
| ADD* | 38 | 59 | 42 | 19 | 31 | 44 | 62 | 63 | 43 | 39 | 52 | 60 | |

Table 4.1: Data Sample (Drawing Frequency Counts)

Table 4.1 illustrates the ADD frequency counts for each of our twelve participants. Each column represents a participant, numbered according to the participant's ID. Each row represents one or more ADD codes in sequence (e.g. ADD-B indicates the number of boxes drawn for a participant, while ADD-BA indicates the number of boxes with arrows drawn). A sequence of ADD codes occurs when subsequent drawings are linked within a single gesture. Data summaries allow the investigator to see clear data trends, by code and by participant. The bottom row is a consistency check that provides the total number of ADD codes that arose from a participant. One may note that the values do not align for participant 5: the consistency checker picked up unintentional ADD codes from outside the transcript region. This is the type of result that merits further investigation.

Table 4.2 illustrates the MEAN *duration* counts. Often, the researcher will investigate not only the counts, but what the counts mean in the context of the participant, the study, or other code counts, durations, or proportions. For example, how long, on average, is the box gesture? Or what proportion of added elements is text? The researcher may examine how the proportion of counts corresponds to the proportion of duration.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MEAN-S-E(* | 0:01:05 | 0:01:00 | 0:00:00 | 0:00:21 | 0:04:01 | 0:02:58 | 0:00:03 | 0:01:38 | 0:01:25 | 0:00:12 | 0:00:26 | 0:00:16 | 0:13:25 |
| MEAN-S-C(* | 0:02:59 | 0:01:05 | 0:02:09 | 0:00:27 | 0:01:51 | 0:00:33 | 0:02:47 | 0:01:28 | 0:04:12 | 0:00:41 | 0:00:40 | 0:05:49 | 0:24:41 |
| MEAN-S-W(* | 0:00:52 | 0:00:54 | 0:00:31 | 0:00:00 | 0:00:11 | 0:00:12 | 0:00:01 | 0:00:11 | 0:00:00 | 0:00:14 | 0:00:00 | 0:00:37 | 0:03:43 |
| MEAN-F-E(* | 0:03:42 | 0:06:22 | 0:01:13 | 0:01:15 | 0:01:54 | 0:01:19 | 0:03:09 | 0:01:23 | 0:06:36 | 0:03:52 | 0:01:09 | 0:01:50 | 0:33:44 |
| MEAN-F-C(* | 0:03:12 | 0:02:12 | 0:02:57 | 0:00:45 | 0:02:07 | 0:06:52 | 0:06:26 | 0:01:23 | 0:03:34 | 0:06:12 | 0:01:16 | 0:06:19 | 0:43:15 |
| MEAN-F-SN(* | 0:01:41 | 0:03:16 | 0:02:10 | 0:00:00 | 0:00:00 | 0:00:00 | 0:00:24 | 0:00:13 | 0:00:00 | 0:00:12 | 0:01:01 | 0:00:12 | 0:09:09 |
| MEAN-A-P(* | 0:01:20 | 0:00:35 | 0:00:05 | 0:03:03 | 0:00:28 | 0:00:17 | 0:00:54 | 0:01:18 | 0:01:19 | 0:00:15 | 0:01:25 | 0:01:23 | 0:12:22 |
| SUM | 0:14:51 | 0:15:24 | 0:09:05 | 0:05:51 | 0:10:32 | 0:12:11 | 0:13:44 | 0:07:34 | 0:17:06 | 0:11:38 | 0:05:57 | 0:16:26 | 2:20:19 |

Table 4.2: Data Sample (Meaning Duration Counts)

**Drawing-Speaking Relationship Matrices**

In the stage of analysis just described, we examined the drawing and speaking codes in terms of counts, durations, and proportions within participants and across the entire study. In the next stage, to determine how the drawing and speaking codes relate, we built a time-ordered relationship matrix (a derivation of our time-ordered matrix), and then built individual relationship frequency matrices 1) in order to demonstrate the occurrence of dialogue (MEAN) that enriches given diagrammatic material (ADD) on a whiteboard, and 2) in order to list relationships and their properties. We believed that the combination of core activities would indicate the link between the explanation activities and knowledge concepts.

To build the matrices, we used the following process. While coding, we assigned a relationship ID to a set of codes if there was a correspondence between a participant's dialogue and what they drew on the whiteboard (in Figure 4.2, the relationship IDs are the unique letters that follow the add and mean codes, 'a', 'b', 'bb'). The next stage was to illustrate the relationships visually to identify a topology of relationships. With a single click the investigator is able to toggle the colour-coding of the time-ordered matrix between an illustration of relationships (Figure 4.12) and an illustration of code categories. Figure 4.12 shows four different kinds of relationships between ADD and MEAN codes:

loners[23] (green colouring), 1-1[24] relationships (aqua colouring), 1:2..* & 2..*:1 relationships (grey colouring), and 2..*:2..* relationships (no colouring).

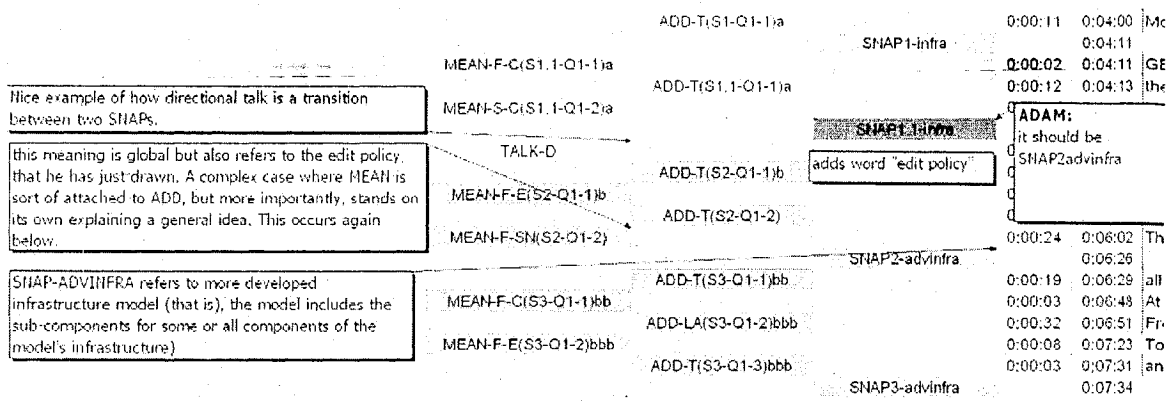


Figure 4.12: Visualizing Relationships (Time-ordered Relationship Matrix)

Next, we formulated several hypotheses based on these relationships and built displays to examine these hypotheses based on relationship types and their properties. We built displays for each participant and a summary of the entire study for each set of relationship-frequency matrices. We built matrices to count the total number of relationships between ADD and MEAN codes (each relationship within a 1:2..* and 2..*:1 is counted once). We built matrices to examine the relationship between single ADD codes and multiple MEAN codes. Then, we built matrices to examine the typical 1-1 ADD-MEAN relationships. We built a list of the loner 1:2, 1:3, 1:4, and 2:1 relationships.

[23] Drawn elements with no corresponding discussion, or discussion of system meaning with no corresponding drawings

[24] The MEAN code precedes the ADD in the relationship, i.e. 1:2 means one MEAN corresponds to two ADD codes

| 1:1 Sum | MEAN-S-E | MEAN-S-C | MEAN-S-W | MEAN-F-E | MEAN-F-C | MEAN-F-SN | MEAN-A-P | SUM |
|---|---|---|---|---|---|---|---|---|
| ADD-B | 31 | 32 | 3 | 34 | 40 | 4 | 10 | **154** |
| ADD-L | 1 | 0 | 0 | 3 | 1 | 0 | 0 | **5** |
| ADD-T | 13 | 26 | 1 | 40 | 34 | 0 | 8 | **122** |
| ADD-A | 1 | 0 | 0 | 0 | 2 | 1 | 1 | **5** |
| ADD-BA | 0 | 3 | 0 | 3 | 3 | 1 | 1 | **11** |
| ADD-TB | 0 | 1 | 1 | 1 | 0 | 0 | 1 | **4** |
| ADD-LB | 0 | 2 | 1 | 1 | 1 | 1 | 0 | **6** |
| ADD-AT | 1 | 3 | 0 | 3 | 3 | 1 | 0 | **11** |
| ADD-C | 0 | 0 | 1 | 2 | 0 | 0 | 0 | **3** |
| ADD-TL | 0 | 0 | 0 | 1 | 0 | 1 | 1 | **3** |
| ADD-LA | 0 | 0 | 0 | 0 | 1 | 0 | 0 | **1** |
| ADD-TLA | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **1** |
| **SUM** | **47** | **67** | **7** | **89** | **85** | **9** | **22** | **326** |

Table 4.3 : Frequency of 1:1 relationships between MEAN and ADD codes

Table 4.3 illustrates the frequency of 1:1 relationships between MEAN and ADD codes. We can see a clear *pattern* from the figure: *text gestures are complemented with a description of software functionality*. *Box gestures*, on the other hand, *are regular for both structure and function* (with slightly greater support for functionality). These are overly simplistic claims, but they illustrate a typical starting point of analysis: find a trend, devise a hypothesis, compare using multiple sources, generate a proposition with corresponding evidence. In practice, further analysis and interpretation is required to produce a general claim.

**Drawing-Speaking-Snapshot Relationship Charts**

At this stage of analysis, we built upon the individual relationships between the drawing and speaking codes through an examination of snapshots. We were interested in the role drawing and speaking play in the generation of a snapshot. To examine this relationship, we built numerous charts illustrating the cumulative frequency or duration of single or multiple codes within an individual snapshot or within the entire explanation as a function of time. We also examined the ratio. We will provide data samples to highlight the types of charts we built.

We built charts to *note relationships among codes*, and to *find patterns* in the data. Our primary rationale for building charts – typical of qualitative research – is to facilitate working at a higher level of abstraction. We can see patterns in a chart that would be difficult to see from the original or coded data. As our data samples will show, charts indicate trends in the relationships among codes for the entire explanation and within

individual snapshots. These trends become a part of the trail of evidence the researcher uses to examine candidate hypotheses and to generate a theory.

To build the charts, we used the following process. As mentioned earlier, while coding, we assigned a unique identifier to each drawing and speaking code instance. In Figure 4.2, the unique ID may be found in brackets, i.e. (S1.1-Q1-1). The unique identifier includes the snapshot ID (the snapshot to which the drawing or dialogue corresponds), the question ID (the question in which the drawing or dialogue occurs), and a numerical ID that increments by one for each subsequent code within a snapshot.

Manual maintenance of code identifiers became a significant challenge and demanded a programmatic solution. The identifier is built according to code location and, where relevant, relationship. All drawing and speaking codes follow a question code and precede a snapshot code, thus permitting an automated pass of the data. This automated pass is not, however, sufficient to take account of drawing or dialogue codes that correspond to another snapshot. To resolve this issue, we use the relationship ID to determine the correspondence between a drawing or dialogue code and an earlier snapshot – the earliest snapshot within a relationship is used as the snapshot ID for all drawing or dialogue codes within that relationship. We performed a full pass of the data to verify that this solution was accurate and to evaluate drawing and dialogue relationships. We programmatically generated all charts according to the unique identifier. To generate thousands of charts in Excel turned out to be a difficult programming challenge.

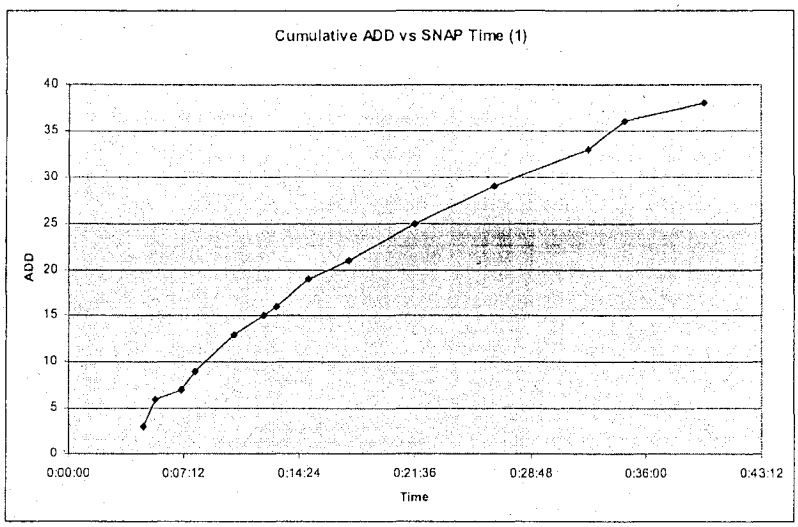


Figure 4.13: Cumulative ADD - Entire Explanation - Frequency - Single Plot

Figure 4.13 illustrates the cumulative frequency of ADD codes as a function of time for the entire explanation of our first participant. Each point on the chart represents a snapshot. This type of chart allows us to examine the relationship between drawing codes and the entire explanation. Figure 4.13 illustrates a trend of a close-to-uniform accumulation of drawings for the entire explanation, with a slight increase in the rate of additions in the first fifteen minutes.

We were interested in the observation of drawing and dialogue codes within a particular snapshot as opposed to the entire explanation. Such observation would allow us to examine events at a lower level and to *make contrasts and comparisons* across groups of similar snapshots (e.g. by snapshot category). We were also interested in the observation of both drawing and speaking codes within one chart in order to examine the interplay between the codes over time.

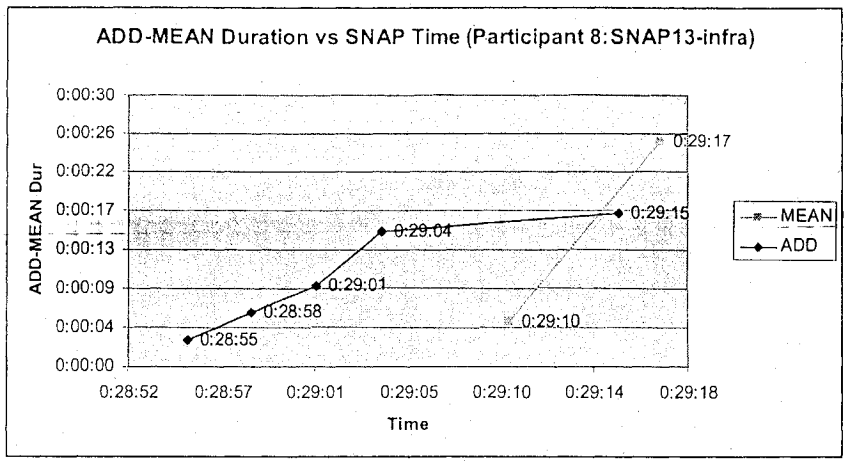


Figure 4.14: Cumulative ADD and MEAN duration in an example infrastructure snapshot. Each point is an ADD or MEAN event.

Figure 4.14 illustrates the cumulative duration of both ADD and MEAN codes as a function of time within the thirteenth snapshot for participant 8. Each point on the chart represents an ADD or MEAN code. Figure 4.14 shows the dominance of ADD codes at the start of the infra snapshot, with supplementary meaning to provide complementary insight.

We were also interested in the proportion of ADD and MEAN codes (and later TALK codes) within SNAP Time. Figure 4.15 illustrates the ratio of ADD and MEAN within an entire explanation. Each bar represents an individual snapshot. The reason each bar does not total 100% is accounted for by the absence of TALK codes within the chart. Figure 4.15 indicates a possible lead: there is a preponderance of ADD codes in the first half of the explanation (the first fifteen-minute segment), and a preponderance of MEAN codes in the second half of the explanation.

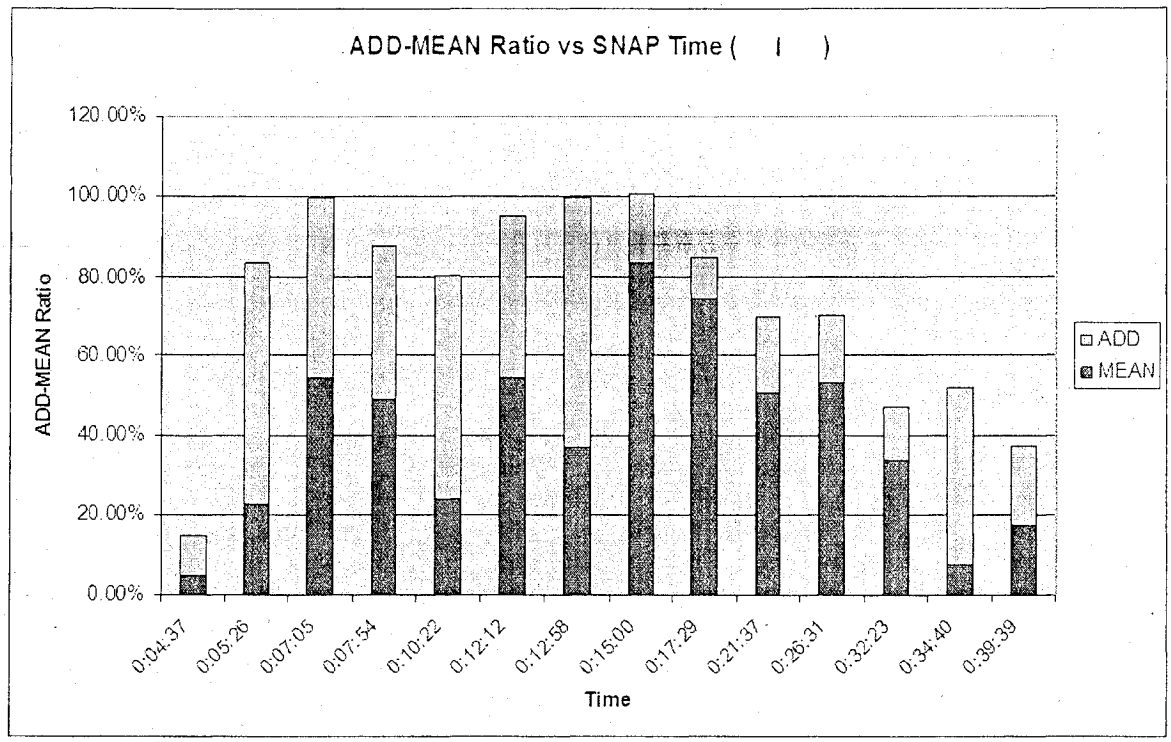


Figure 4.15: Percentages of ADD and MEAN codes in an example complete session. Each bar is a snapshot.

### Snapshot Matrices

At this stage of analysis, we examine how drawing and speaking codes relate to snapshots at a more general level. We built several matrices to examine how the frequency and duration of different categories of ADD, MEAN and TALK codes correspond to different categories of SNAP codes. Aside from snapshot category, we also examine frequency and duration by increment (first snapshot, nth snapshot, etc.), and by zone (for which the researcher specifies two values: the number of codes before and the number of codes after to search against). Aside from *counting*, matrices support the examination of other trends and patterns. In this section, we provide a sample of our snapshot matrix displays.

| SUM | infra | advinfra | func | exam | L | S | complete | weak | SUM |
|---|---|---|---|---|---|---|---|---|---|
| ADD-B | 97 | 27 | 25 | 7 | 35 | 3 | 52 | 6 | **252** |
| ADD-L | 5 | 1 | 1 | 0 | 0 | 0 | 3 | 0 | **10** |
| ADD-T | 55 | 20 | 64 | 9 | 52 | 3 | 26 | 9 | **238** |
| ADD-A | 9 | 10 | 11 | 1 | 12 | 0 | 11 | 1 | **55** |
| ADD-BA | 1 | 1 | 5 | 0 | 5 | 0 | 3 | 0 | **15** |
| ADD-TB | 1 | 0 | 3 | 0 | 1 | 0 | 0 | 0 | **5** |
| ADD-LB | 1 | 0 | 3 | 0 | 0 | 0 | 1 | 0 | **5** |
| ADD-AT | 0 | 1 | 8 | 1 | 5 | 2 | 9 | 8 | **34** |
| ADD-C | 2 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | **6** |
| ADD-TL | 0 | 0 | 0 | 3 | 0 | 1 | 2 | 0 | **6** |
| ADD-LA | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | **3** |
| ADD-TLA | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | **1** |
| **SUM** | **171** | **62** | **122** | **21** | **111** | **9** | **109** | **25** | **626** |

Table 4.4: ADD vs. SNAP Category Frequency

Table 4.4 illustrates a summary matrix (summation of all participants) for the frequency of ADD categories by SNAP categories. We populated the category-frequency matrix by counting the codes of different categories under every snapshot. The right column and bottom row contain sums. There is a disparity between the sum values in Table 4.1 and Table 4.4. This is to be attributed, as stated in the coding scheme, to the lack of mutual exclusion for SNAP categories. For example, a snapshot can simultaneously be a lateral helper and functional. Therefore, during analysis, we limited our examination to *counting* within mutually exclusive sets (e.g. we did not compare the lateral and functional values).

From Table 4.4, we observe a large proportion of box (40%) and text (38%) diagrammatic elements. This shows that the preferred way to explain software is through *boxes and text.* A deeper look at the data (not shown here) indicates that *boxes represent more structural aspects of software and text represents more functional aspects of software.* Data from duration matrices (also not shown here) supports the dominance of box and text events.

| SUM | infra | advinfra | func | exam | L | S | complete | weak | SUM |
|---|---|---|---|---|---|---|---|---|---|
| MEAN-S-E | 18 | 4 | 6 | 0 | 10 | 2 | 11 | 2 | **53** |
| MEAN-S-C | 26 | 6 | 26 | 0 | 15 | 0 | 7 | 4 | **84** |
| MEAN-S-W | 2 | 3 | 2 | 0 | 1 | 0 | 6 | 1 | **15** |
| MEAN-F-E | 19 | 5 | 26 | 5 | 16 | 5 | 18 | 5 | **99** |
| MEAN-F-C | 7 | 8 | 26 | 3 | 21 | 1 | 18 | 4 | **88** |
| MEAN-F-SN | 1 | 2 | 6 | 2 | 5 | 1 | 8 | 0 | **25** |
| MEAN-A-P | 1 | 1 | 6 | 0 | 5 | 0 | 3 | 0 | **16** |
| **SUM** | **74** | **29** | **98** | **10** | **73** | **9** | **71** | **16** | **339** |

Table 4.5: MEAN vs. SNAP Category Frequency

Table 4.5 similarly illustrates frequencies of MEAN events versus snapshot category. The data illustrates that functional meaning occurs more frequently than structural meaning. *Functional meaning is critical and dominant in the creation of the majority of snapshot categories*. The exception to this claim is the infrastructure snapshot, which is largely composed of structural meaning. Structural meaning plays a minor role in the composition of lateral and complete snapshots. The small frequency of meaning in example snapshots is entirely composed of functional meaning. Table 4.6 illustrates the duration breakdown for the entire study. One interesting point is that precisely the same proportion of drawing took place in the pilot study – 25%.

| | | |
|---|---|---|
| Total Study | 5:59:39 | |
| Drawing | 1:30:32 | 25% |
| Meaning | 2:20:19 | 39% |
| Talking | 2:06:21 | 35% |
| Noise/Other | 0:02:27 | 1% |

Table 4.6: Overall Study Duration and Proportion

We looked at both frequency and duration (e.g. Table 4.7) matrices to support our findings. When drawing, the participant drew boxes for 34:10 (38%) and text for 33:01 (36%). When providing meaning, the participant provided structural meaning for 41:49 (30%) and functional meaning for 1:26:08 (61%).

| 12 | SNAP1-inf | SNAP2-ad | SNAP3-L- | SNAP4-co | SNAP5-inf | SNAP6-co | SNAP7-fu | SNAP8-ad | SNAP9-inf | SNAP10-fi | SNAP11-ir | SNAP12-fi | SNAPx | SUM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MEAN-S-E | | 0:00:05 | 0:00:02 | 0:00:09 | | | | | | | | | | **0:00:16** |
| MEAN-S-C | 0:00:04 | | 0:00:09 | | 0:00:11 | | 0:01:44 | 0:00:07 | 0:00:58 | | | 0:01:56 | | **0:05:09** |
| MEAN-S-W | | 0:00:10 | | | | | | 0:00:27 | | | | | | **0:00:37** |
| MEAN-F-E | | | | 0:00:35 | 0:00:11 | | | | | | 0:00:15 | | | **0:01:01** |
| MEAN-F-C | | | | | | | 0:00:29 | 0:00:22 | 0:00:21 | | 0:00:57 | 0:01:21 | | **0:03:30** |
| MEAN-F-SN | | | | | | | | | | 0:00:12 | | | | **0:00:12** |
| MEAN-A-P | | | | | | | | | | | | 0:01:11 | | **0:01:11** |
| **SUM** | **0:00:04** | **0:00:15** | **0:00:11** | **0:00:44** | **0:00:22** | **0:00:00** | **0:02:13** | **0:00:56** | **0:01:19** | **0:00:12** | **0:01:12** | **0:04:28** | **0:00:00** | **0:11:56** |

Table 4.7: MEAN vs. SNAP Increment Duration

Table 4.7 illustrates the duration of mean categories across individual snapshots for participant 12. This type of matrix provides a context for other analyses. For example, we can relate how the matrix in Table 4.2 is filled, or we can deepen our understanding of the composition for a chart, such as Figure 4.14, and so forth. In Table 4.7, we see a new column, entitled SNAPx. A participant would sometimes provide drawing or dialogue after the last snapshot but before the end of the explanation. We used SNAPx as a hidden code to catch all codes which did not correspond to a particular snapshot at the end of an explanation.

| SUM | infra | advinfra | func | exam | L | S | complete | weak | SUM |
|---|---|---|---|---|---|---|---|---|---|
| TALK-C-SE | 0 | 1 | 1 | 0 | 0 | 0 | 2 | 0 | **4** |
| TALK-C-OE | 3 | 1 | 3 | 0 | 0 | 0 | 4 | 0 | **11** |
| TALK-JT | 4 | 3 | 0 | 0 | 1 | 0 | 2 | 0 | **10** |
| TALK-D | 8 | 6 | 1 | 3 | 2 | 0 | 5 | 1 | **26** |
| TALKx | 35 | 10 | 22 | 1 | 27 | 1 | 39 | 7 | **142** |
| **SUM** | **50** | **21** | **27** | **4** | **30** | **1** | **52** | **8** | **25** |

Table 4.8: TALK vs. SNAP Zone Frequency

Table 4.8 may appear similar to Table 4.4, but there is a significant difference. Table 4.4 counts code categories within a snapshot of a particular category; that is, counting is based on the snapshot identifier for the drawing or dialogue code. In contrast, Table 4.8 counts code categories within the *zones* of snapshots of a particular category; that is, the user specifies a zone (a.k.a. a window) of interest (e.g. x codes above, y codes below, where x and y are non-negative integers that specify the number of codes above or below the snapshot code to search), and we count codes by category if they exist within the snapshot zone. Zones can overlap; if we specify 300 codes above and 300 codes below, then each zone will include the entire explanation, and the matrix will include large counts that are difficult to interpret, though such zones are not the intended use of this matrix. The rationale for this type of display is to determine the categories of codes that immediately precede or follow snapshots of a particular category. We provide zones so the user can specify what is meant by 'immediately precede or follow.'

**Snapshot Event State Network**

We built matrices and charts to understand snapshots in terms of their relationship with other codes. For the next analytic step, we built displays to examine the relationship

between snapshots. To achieve this, we built event-state networks for the entire explanation for each participant. This type of display shows us how snapshots function at a high level of abstraction – within an entire explanation, and with other snapshots. While we built networks, we gained insight into the overall explanation process for each participant, but we also encountered strange cases that merited further investigation of the source data. We began to appreciate the dual role of displays: deeper insight and data integrity.

To build the networks, we first settled on a notation that was a derivation of similar snapshot diagrams we had drawn over the course of analysis, for example, in the discovery of lateral helpers. We demonstrate our notation in Figure 4.16.

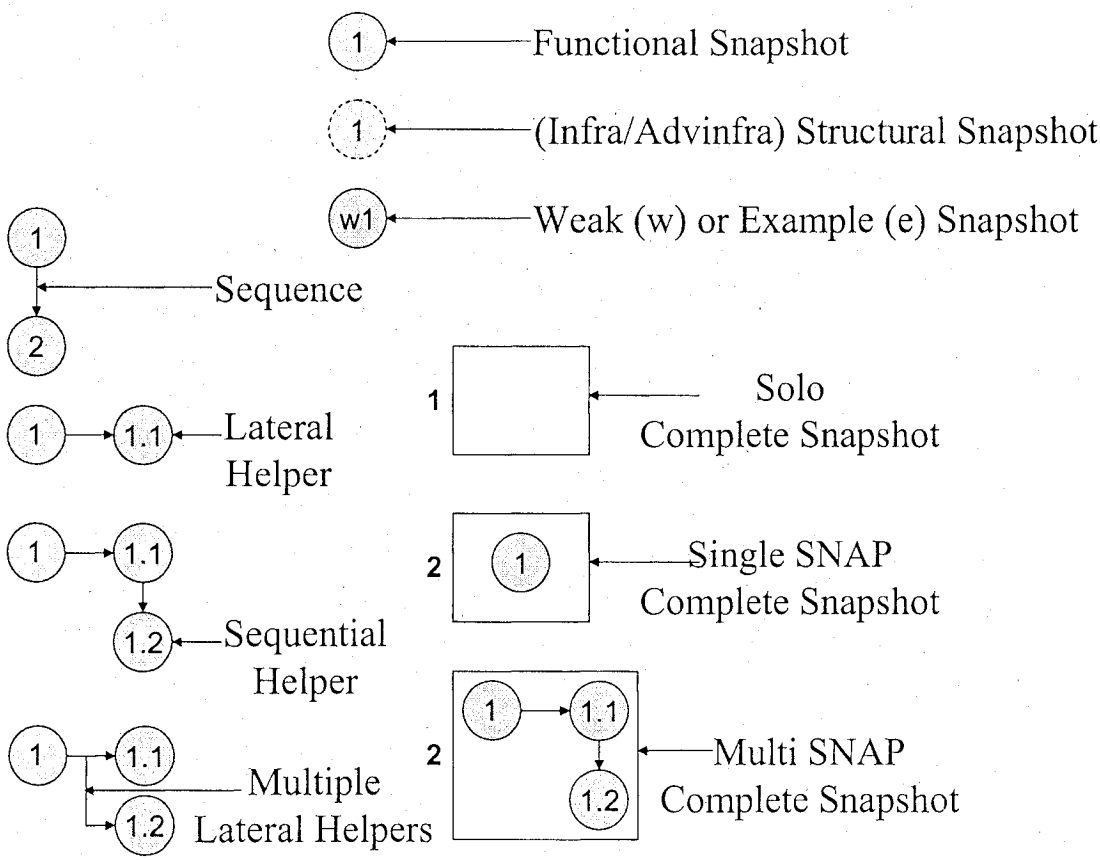


Figure 4.16: Snapshot Event Network Legend

Once we had the notation in place, we built the networks for each participant in order to understand the snapshot relationships. First, we built a snapshot-event matrix (Figure 4.17). This type of matrix is simple to build, but provides core information in one place.

We coloured the matrix according to snapshot category. Then, we followed the notation and built the networks according to the types and sequence as presented in the event matrix.

| Participant 1 | |
|---|---|
| Q1 | 0:03:42 |
| SNAP1-infra | 0:04:37 |
| SNAP2-advinfra | 0:05:26 |
| SNAP2.1-L-func | 0:07:05 |
| Q2 | 0:07:54 |
| [redacted] | 0:07:54 |
| SNAP4-func | 0:10:22 |
| SNAP4.1-L-func | 0:12:12 |
| SNAP4.2-infra | 0:12:58 |
| [redacted] | 0:15:00 |
| [redacted] | 0:17:29 |
| Q4 | 0:20:15 |
| SNAP5-func | 0:21:37 |
| [redacted] | 0:26:31 |
| Q5 | 0:28:10 |
| Q6 | 0:32:23 |
| [redacted] | 0:32:23 |
| [redacted] | 0:34:40 |
| Q10 | 0:34:40 |
| [redacted] | 0:39:39 |
| Q11 | 0:40:11 |

| Participant 2 | |
|---|---|
| Q1 | 0:03:24 |
| SNAP1-infra | 0:04:11 |
| SNAP1.1-infra | 0:04:33 |
| SNAP2-advinfra | 0:06:26 |
| SNAP3-advinfra | 0:07:34 |
| SNAP3.1-L-func | 0:09:05 |
| SNAP3.2-L-func | 0:10:24 |
| SNAP3.3-S-func | 0:12:02 |
| SNAP3.3.1-L-func | 0:14:26 |
| [redacted] | 0:15:40 |
| SNAP3.4-L-func | 0:17:58 |
| Q2 | 0:17:58 |
| SNAP3.3.2-L-func | 0:22:03 |
| SNAP3.4.1-infra | 0:23:09 |
| SNAP3.4.2-func | 0:28:29 |
| SNAP5-exam | 0:31:48 |
| SNAP5.1-S-exam | 0:33:16 |
| Q10 | 0:38:37 |
| SNAP5.2-exam | 0:38:37 |

| Participant 3 | |
|---|---|
| Q1 | 0:00:05 |
| SNAP1-infra | 0:01:19 |
| [redacted] | 0:02:25 |
| Q2 | 0:02:28 |
| SNAP3-infra | 0:05:11 |
| SNAP4-advinfra | 0:07:26 |
| SNAP5-advinfra | 0:09:58 |
| Q3 | 0:14:30 |
| [redacted] | 0:14:30 |
| Q4 | 0:14:55 |
| Q4 | 0:17:29 |
| Q5 | 0:19:30 |
| Q6 | 0:20:55 |
| Q7 | 0:22:10 |
| Q7 | 0:23:35 |
| Q7 | 0:24:11 |
| Q8 | 0:25:28 |
| Q10 | 0:28:00 |
| END | 0:30:07 |

| Participant 4 | |
|---|---|
| Q1 | 0:04:55 |
| SNAP1-infra | 0:07:54 |
| [redacted] | 0:10:39 |
| Q2 | 0:13:15 |
| [redacted] | 0:13:15 |
| [redacted] | 0:14:53 |
| Q3 | 0:15:10 |
| Q4 | 0:17:23 |
| Q5 | 0:18:54 |
| Q6 | 0:22:14 |
| Q7 | 0:23:01 |
| Q7 | 0:24:22 |
| Q8 | 0:25:24 |
| Q9 | 0:26:36 |
| Q10 | 0:28:48 |
| END | 0:29:08 |

Figure 4.17: Snapshot Event Matrix

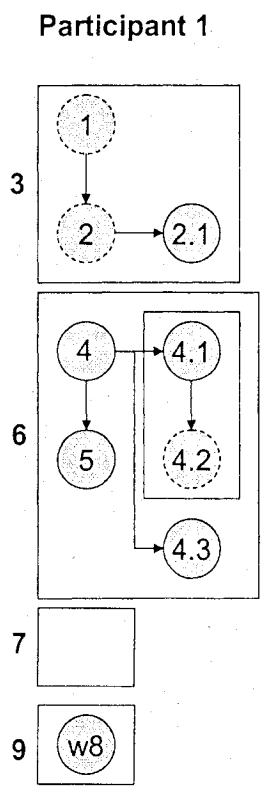


Figure 4.18: Snapshot Event Network Data Sample

In Figure 4.18, an example network for participant 1, we see five complete snapshots, one of which is nested. The first complete snapshot represents a typical structure; we examine the topology of snapshot-event networks in the results of our interpretation phase, discussed in next chapter. We also see cases of solo complete snapshots, and complete snapshots based on weak explanation. We can also observe four core concepts (snapshots 1, 2, 4, 5 respectively), and how they fall into two major explanations (complete snapshots 3 and 6 respectively). We see structural snapshots as the foundation for the explanation (the first two snapshots), and the regularity of functional snapshots as the first snapshot in a lateral branch.

These networks are rich for exploration, and in the next chapter we reveal some of our findings from the comparison of event networks across multiple participants. We suggest that as future work it would be a good idea to compare the networks built by multiple researchers; this would help ensure the building process is unbiased and representative of the data. This is a form of inter-rater reliability checking.

**Other types of displays**

We produced matrices on an as-needed basis as one step in the construction of an analytic base in following a trail of evidence. At various stages of analysis, we built displays that did not work. However, the discarded displays might be relevant for other qualitative inquiries, and in fact may be useful to our inquiry in the future. In addition, we had unfulfilled plans to build other displays. For completeness, the following describes the displays that did not pan out.

We built a *role-ordered matrix* to examine the effects of role and experience on the production of snapshots. However, we built this display very early (before we revised the coding scheme), and we could not find any trends in the data. Perhaps the more detailed insight into snapshots would lead us to interpret the data differently, but we have, at present, no intention of returning to this line of inquiry.

As part of the early stages of content analysis, we performed *word frequency analysis*. We built token frequency tables for each participant and for the entire study, and sorted the tables by frequency and alphabetically. Our goal with this line of inquiry was to more accurately follow the grounded theory approach and to produce a model of the user's

knowledge that we intended to align with our snapshot model. The integration of the two models was meant to provide insight into the role of the snapshot in the developer's cognitive model. We abandoned this line of research because it was unclear how we should integrate the two models and whether we were yet able to describe the cognitive model. Thus describes how we proceeded with topical analysis; the next stage of our research, as described in the next section, was category analysis.

## 4.4 Category Analysis

To perform category analysis, we examined each snapshot of a particular category. We looked for evidence in the data that objectively satisfied our research questions and drafted the answers in narrative form; thus, the steps of our snapshot category analysis produce a snapshot *story*. The following describes how we produced the narrative stories.

While coding, we determined that each snapshot category had unique properties that may yield insight into snapshots as a whole. We built storyboards for each snapshot of a particular category. Appendix B contains example storyboards of two snapshots for each category. We then analysed the storyboards to gain insight into *commonalities and differences* within and among snapshot categories, the *conditions* that contribute to the existence and occurrence of snapshots, and participant *strategies*. We constructed each storyboard to contain the following information:

- We found *conditions* in the Drawing-Speaking Independent and Relationship Matrices: ADD / MEAN frequency, duration, sequence, relationships, loners, introductory summaries and concluding talks.
- We found *context* in the contextual TALK codes and Time-Ordered Matrix:
    - TALK frequency
    - Duration
    - Self-evaluation
    - Objective-evaluation
    - Researcher's directed questions
    - Time constraints, and
    - Noise.

- We measured the explanation and snapshot times, the number of drawn elements, the corrections, and the length of pause immediately following the snapshot event.
- We found the participant *strategy* to be revealed in the explanation sequence, element order and the rationale for participant explanation.
- We were interested in the *consequence* as quality of the message in terms of information density, but the investigation was unsuccessful. Our data did not yield insight into cognitive effect and our interpretation of the message quality is subjective. That said, future research could reveal the consequences of snapshots to software understanding.

From the conditions, context, and strategies, we wrote a narrative description of a composite (not case) snapshot for each snapshot category. We also revised the level of generality of claims by analysing the story associated with other snapshots of the same category. The contextual mediator in the story was the whiteboard, which participants used to graphically construct software models. Our stories included supporting data of conditions, relationships between codes and directionality. To generalize, the actor in our story was "the participant," though we examined many participants to produce the story; "the participant" means "the participants in general". In Section 5.3, we present a uniform Snapshot Theory derived from the convergence of snapshot features across snapshot stories.

## 4.5 Concluding remarks on study design

The value of a qualitative data study rests heavily on the quality of the data analysis. In this chapter, we have outlined the design and execution for a study of the temporal details of software. We built roughly 8000 displays, and in this chapter our aim has been to provide a cross-section of those displays with data samples and our rationale. In the next chapter, we provide the results we derived from the displays: propositions tied to a trail of evidence, hypotheses grounded in the data, explanations regarding the phenomenon under study, and areas for future study.

# Chapter 5 Snapshot Theory

We used the data analysis steps described in the last chapter to construct the results of our qualitative study of snapshots in software explanation. In this chapter, we report the results, which we call Snapshot Theory.

We use three main approaches to describe Snapshot Theory. The first is a simple table of snapshot categories containing a set of typical *indicators* to be used for identifying snapshots (Section 5.1). Next (Section 5.2), we provide a more extensive description of the snapshot categories using *snapshot stories*, which we derived from storyboards (see Appendix B for storyboard examples). In Section 5.3, we summarize Snapshot Theory, integrating the stories and providing some additional discussion.

## 5.1 Typical indicators for identifying snapshots

The building blocks of discourse structure identified in Snapshot Theory are the snapshots themselves. Researchers require a way to reliably identify each category of snapshot. A key result of our analysis, therefore, has been to identify a simple set of 'indicators' that can be used by different people to consistently identify and categorize the same set of snapshots.

Table 5.1 summarizes these indicators for the identification of snapshots; this table builds upon the coding scheme described in Section 4.3.2. The first column lists snapshot category. The second column provides a one-line description for the snapshot category. A coder examining some discourse can use the third column as a reference of typical indicators for the category of snapshot currently under observation. This table is presented first to help the reader understand the snapshot categories; subsequent sections provide much more detail about each category.

| Snapshot Category | Definition | Typical indicators in the discourse leading up to this category of snapshot. |
|---|---|---|
| Infrastructure | General overview that contains fundamental architectural details | 1) Presents core structural components<br>2) Closes with an overarching statement<br>3) Explicit meaning is tied to diagrammatic elements<br>4) More drawing than speaking occurs. |
| Advanced Infrastructure | Discussion of deeper structure with detailed focus on a specific structural component | 1) Begins with and closes with an evaluation of structure<br>2) Focus is on a specific structural component<br>3) Explicit meaning is tied to diagrammatic elements<br>4) More drawing than speaking occurs. |
| Functional | Detailed discussion of processes or mechanics of a specific element | 1) Primarily text<br>2) Explicit ties to prior snapshots<br>3) Significant portion of diagrammatic elements without meaning and meaning that does not link to diagrammatic elements<br>4) More speaking than drawing. |
| Lateral | Divergent discussion that questions the role of a specific diagrammatic element | 1) Identifies specific diagrammatic element and questions the role of the element<br>2) Directional purpose established<br>3) Ties lateral discussion to infrastructure<br>4) Significant portion of diagrammatic elements without meaning and meaning that does not link to diagrammatic elements. |
| Example | Discussion of the applied usage of a diagrammatic element | 1) Begins with an utterance that contains a question<br>2) Answers question with an example<br>3) Answer may include structure and function<br>4) Ends with direction to other practical sub-questions. |
| Weak | Discussion which lacks structure and/or functional elements necessary for comprehension | 1) Starts with self-evaluation<br>2) Estimate of what current model should look like<br>3) Premature abandonment of functional discussion<br>4) Primarily general discussion and self-evaluation<br>5) Seldom links to prior snapshots. |
| Complete | Global discussion that combines prior snapshots | 1) Starts with infrastructure snapshot<br>2) If infrastructure is insufficient, then continues with either advanced infrastructure or functional<br>3) Closes with overarching statement. |

Table 5.1: Typical indicators for the identification of snapshots

## 5.2 Snapshot Stories

This section adds more detail to the analysis of the snapshot categories presented above. We describe the snapshot categories in terms of conditions, context and the participants' behavioural strategies. The analysis allows us to answer our research questions from Section 1.1; we address, for example, snapshot composition and structure.

Each of the following sub-sections contains a *snapshot story* for a snapshot category. We remind the reader that when we refer to explanation in the following, we mean both drawing and dialogue events. Furthermore, some dialogue events consist of the participant providing meaning.

### 5.2.1 Infrastructure Snapshot

The infrastructure snapshot precedes other snapshots in the development of a complete snapshot. The implication of infrastructure snapshots is that basic structure is critical for software developers' understanding of a software model.

In all cases, the participant produces an infrastructure snapshot as the first snapshot. Table 5.2 illustrates that in 10 of 12 cases, the participant exhibits a higher frequency of ADD codes than MEAN codes before the first snapshot.

| Participant | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | AVG |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD | 3 | 1 | 2 | 4 | 0 | 2 | 2 | 9 | 13 | 4 | 8 | 7 | 4.6 |
| MEAN | 3 | 0 | 0 | 3 | 0 | 1 | 3 | 0 | 3 | 1 | 1 | 5 | 1.7 |

Table 5.2: Comparison of ADD and MEAN code frequency before first snapshot

Within the infrastructure snapshot, the participant spends more time drawing the basic software architecture than time providing the meaning of the architecture. The participant creating the infrastructure snapshot draws more and provides more meaning than in other snapshot categories.

The participant starts the discourse leading up to an infrastructure snapshot with a general overview of the topic to be presented. The participant draws the main structure in vertical or horizontal form. The participant may omit or skip diagrammatic elements in order to emphasize essential details. The participant may not discuss the links between components. The explanation focuses on fundamental architectural details – the basics or the core structural components. The explanation may conclude with a summary or

overarching statement (*"This is a basis of our diagrams"*). Or, the participant proceeds without summary; in such cases, the participant may demonstrate that the topic is self-explanatory with a nod that suggests 'this is clear.'

When the participant speaks generally, in order to organize for the upcoming explanation, he or she directs the attention of the audience to the next topic with phrases such as "*Let me now explain why I have this box crossing both*" and "*Now ... what's happened with ...?*"

The participant reaches the infrastructure snapshot after roughly two minutes of explanation, after drawing 3 to 6 boxes or 2 to 5 textual elements on the whiteboard. The participant has either drawn boxes with labels or text with no corresponding shapes. For the most part, the participant explicitly describes the meaning of the diagrammatic elements. To not explain a diagrammatic element is rare and, likewise, to explain the software's meaning without corresponding drawings is also rare.

### 5.2.2 Advanced Infrastructure Snapshot

In the development of a complete snapshot, the advanced infrastructure snapshot builds atop the infrastructure snapshot. The implication of advanced infrastructure snapshots is that different levels of abstraction and detail are necessary for software developers' understanding of a software model. Though not as prevalent as the infrastructure snapshot, the advanced infrastructure snapshot has its place. In many cases, the basic software architecture provided with the infrastructure snapshot will not suffice and software developers require deeper structure in order to understand a software model.

Within the advanced infrastructure snapshot, the participant spends more time drawing the deeper software architecture than time providing the meaning of the architecture. Between the advanced infrastructure snapshot and the infrastructure snapshot categories, the participant provides less meaning and draws less frequently for the advanced category; the reason for this difference is that the participant focuses on a particular structural element. Between the advanced infrastructure snapshot and weak snapshot categories, the participant provides more meaning and draws more frequently for the advanced infrastructure category.

The advanced infrastructure snapshot also has more diagrammatic elements without meaning and more meaning that does not link to diagrammatic elements than the infrastructure snapshot.

The discourse leading to the advanced infrastructure starts with an evaluation of the structure (e.g. "... *I've found that trying to make this huge thing with all these little boxes just turns out [to be] incomprehensible*"). The participant then draws a number of boxes and textual elements in the model. The participant links diagrammatic elements or meaning to the prior infrastructure snapshot. The explanation concludes with an evaluation of the whole structure.

The participant reaches the advanced infrastructure snapshot after an average of 50 seconds of explanation. The advanced infrastructure is proportionally one-third of the time of the infrastructure snapshot. For advanced infrastructure snapshots, the use of the whiteboard is more frequent and lengthy than the provision of meaning. When the participant speaks generally, he or she provides an objective evaluation of the more developed model.

### 5.2.3 Functional Snapshot

The participant produces the functional snapshot through the selection of a specific diagrammatic element, the drawing of text, the functional meaning of this text, and the general discussion that frequently follows the functional meaning of a functional snapshot.

The participant starts the discourse leading up to a functional snapshot by selecting a diagrammatic element and providing some meaning to it ("*These are plug-ins. Think in terms of plug-ins. They provide some service.*"). The participant then draws the textual elements – roughly two-and-a half times more of these than boxes. In creating the functional snapshot, the participant draws less than is the case for the infrastructure snapshot, but more than all the other snapshot categories. In creating the functional snapshot, the participant spends almost twice as much time providing meaning as drawing – more than is the case in all other snapshot categories – which corresponds to the observed tendency to select textual elements that demand further meaning.

The participant links the present functional snapshot with prior snapshots. Much general discussion supports the functional snapshot. The participant uses functional

discussion for the majority of the lateral snapshots, described in the next section. When the participant speaks generally, he or she provides an objective evaluation of the model.

The participant reaches the functional snapshot after roughly four minutes of explanation, which accounts for the substantial portion of overall functional discussion. Within the functional snapshot, the participant often provides multiple meanings to diagrammatic elements. Furthermore, diagrammatic elements without meaning – and meaning that does not link to diagrammatic elements – occupy more than half of the total duration. These lead us to the observation that *functionality is more complex than structure.*

### 5.2.4 Lateral Snapshot

The participant produces the lateral snapshot as follows:

- selecting a specific diagrammatic element;
- providing functional or structural meaning for the element;
- questioning the role of the element;
- directing explanation flow.

The participant starts the discourse leading to a lateral snapshot by highlighting a diagrammatic element and asking a corresponding question (e.g. "*Coming back to the model, whenever an element changes there are events that get fired; how do we listen to it?*"). The participant then provides directional discussion to establish the purpose of the lateral model. Next, the participant draws more textual elements than boxes and describes the elements' functionality. Sometimes, the participant describes the elements' structures. Among snapshot categories, the participant draws less and provides less meaning than the infrastructure and functional snapshots. Within the lateral category, the participant draws less than he or she provides meaning – a tendency that corresponds to the dominant use of lateral snapshots for functional discussion.

The participant recalls the link between the present lateral snapshot and the prior infrastructure snapshot (e.g. "*and if you remember the previous drawing, we have a C/C++ box. We're just reusing the same box from ...*"). Although divergence is the hallmark of lateral discussion, the participant rarely speaks generally or diverges from the lateral discussion. That is, to create the lateral snapshot, the participant does not provide anecdotes, evaluations or directional discussion.

The participant reaches the lateral snapshot after an average of two minutes of explanation. Among snapshot categories, the lateral snapshots demonstrate a much higher frequency of diagrammatic elements without meaning – and meaning that does not link to diagrammatic elements. There are two possible reasons for this: 1) The participant expects the audience to fill in the gaps; 2) The material is non-essential and the participant believes that gaps will not have a detrimental effect on software understanding.

### 5.2.5 Example Snapshot

The participant produces the example snapshot as follows:

- stating a practical question
- providing detailed functional meaning
- directing the audience's attention to other practical sub-questions.

The participant starts the discourse leading to the example snapshot with a focusing remark and a practical question (e.g. "*Now, we drew this (the list) on the screen. How does that work?*"). The participant then continues with additional questions and answers with an example (e.g. "*The one issue could be how do you get to this, how do you make a query?*"). The participant draws on average one-third the diagrammatic elements for the example snapshot category than would be the case for the advanced infrastructure snapshot category. The participant describes the model's function, with no structural clarification. The example snapshot occasionally ends with an evaluation, but general discussion is rare.

We found the example snapshot to be rare, accounting for only 1/30 of total explanation time. The participant typically reaches the example snapshot after roughly three minutes of explanation and provided the meaning of about half of the diagrammatic elements.

### 5.2.6 Weak Snapshot

The participant produces the weak snapshot when the topic for explanation is outside the bounds of the participant's expertise and when descriptive meaning is replaced with extensive general discussion. The weak snapshot does not adequately address structural meaning and does not provide an overarching description of the software model.

The participant starts the discourse leading to a weak snapshot with a self-evaluation, such as how much information the participant possesses or does not possess in respect to a topic (e.g. *"I didn't have a lot of experience with using [this technology] and seeing exactly how they implemented..."* or *"I am trying to think because it has been some time since I was on this."*). The participant proceeds with an estimate of what the structure of the model should look like through the addition of diagrammatic elements. We use the term "estimate" because the participant qualifies the diagrammatic elements with the word "should." The participant produces typically four weak snapshots per session, half of which are purely functional and half of which have both structural and functional meaning.

The participant provides more structural meaning than functional meaning in terms of duration and more functional meaning than structural meaning in terms of frequency. This leads to 'scattered functional meaning'. Following up on the scattered functional meaning in weak snapshots, we found that the participant prematurely abandons functional discussion in favour of general discussion and self-evaluations that do not adequately fill the gap. Moreover, an overarching description of the model's functionality is missing and the participant does not link the snapshot with prior snapshots.

The participant reaches the weak snapshot after roughly one and a half minutes of explanation. The frequency of diagrammatic elements without meaning and meaning that does not link to diagrammatic elements occupies roughly one half of the overall explanation time for weak snapshots. Among snapshot categories, the weak snapshot occupies a small portion of overall explanation time. The participant often speaks generally and provides ample self-evaluations.

### 5.2.7 Complete Snapshot

The core process in the construction of an explanation is the construction of a complete snapshot. The branches in a snapshot event network (e.g. Figure 4.18) reveal the participant's structural organization of key knowledge units with assigned priority. If the participant assigns priority to too many details or lateral snapshots, it is difficult to produce a complete snapshot. This structure may take linear or non-linear shape. In non-linear shapes, the participant produces branches of lateral functional snapshots. The complete snapshot typically has one of the following structural organizations:

- infrastructure complete snapshot (e.g. Figure 4.18, Snapshot 7, which had predominantly structural discussion)
- structural complete snapshot (e.g. Figure 4.18, Snapshot 3, without the lateral snapshot, 2.1)
- L-complete snapshot (structural complete snapshot with lateral snapshot) (e.g. Figure 4.18, Snapshot 3)
- L-developed complete snapshot (L-complete with other complete snapshots or developed snapshots) (e.g. Figure 4.18, Snapshot 6).

The participant produces a complete snapshot as a combination of prior snapshots from the start of the explanation or since the last complete snapshot. The participant relies on the conditions and contextual variables from prior snapshots to produce the complete snapshot. Therefore, we describe the complete snapshot more in terms of the composition of other snapshots, and less in terms of drawing, meaning and speaking events.

The participant primarily builds the first complete snapshot by relying on infrastructure and advanced infrastructure snapshots. In 4 of 12 cases, the infrastructure was sufficient for a complete snapshot, that is, the participant insists that the structure is sufficient for general understanding. In 3 of 12 cases, the participant produces an additional advanced infrastructure that is then sufficient for a complete snapshot. In the remaining 5 of 12 cases, the participant produces functional snapshots in addition to the structural snapshots; together these are sufficient to constitute a complete snapshot. The participant builds successive complete snapshots with structural snapshots combined with lateral functional snapshots. Of 26 complete snapshots in 12 explanations, the participant used 36 infrastructural, 14 advanced infrastructural and 19 functional snapshots. When the participant builds successive complete snapshots for new questions, they usually do so through the use of infrastructure snapshots.

In general, the participant spends more time drawing than describing the models' meaning. The participant often starts and ends the complete snapshot with a directional discussion (e.g. "*That's the way the system works*"). In 5 of 12 explanations, the participant reached the complete snapshot in roughly 5 minutes; in 7 of 12 explanations, the participant reached the complete snapshot in roughly 15 minutes.

The Snapshot Event Matrix reveals many trends regarding the composition of complete snapshots. *Software explanation begins with and relies upon structure.* Structural snapshots are a uniform requirement in the creation of the first complete snapshot. The first complete snapshot follows a structural snapshot for 9 of 12 participants and then follows a lateral functional snapshot for the remaining 3 of 12 participants. In the 9 of 12 cases, 3 participants produced a complete snapshot after the first snapshot, an infrastructure snapshot and another 3 produced a complete snapshot after the second snapshot – an advanced infrastructure snapshot.

Seven of 12 participants produced a second complete snapshot. Structural snapshots are again a uniform requirement in the creation of the second complete snapshot. Only 1 participant produced a lateral functional snapshot in addition to a structural snapshot to build the second complete snapshot.

Four of 12 participants produced a third complete snapshot. One participant produced a structural snapshot to build the complete snapshot, whereas another participant built a stand-alone complete snapshot and the remaining 2 produced lateral functional snapshots in anticipation of the complete snapshot.

The composition of a complete snapshot consists of a number of other snapshots. To address whether the number of snapshots relate to the integrity and maturity of participant knowledge, we used a threshold of 9 snapshots to group participants into 2 groups of 6 participants (group 1: greater than or equal to 9 snapshots; group 2: less than 9 snapshots). Table 5.3**Error! Reference source not found.** illustrates the frequency of occurrence for categories of snapshots within each group and the proportion of each category expressed as the number of snapshots of a particular category divided by the total snapshots within the group. The frequency of weak snapshots was too low to merit analysis, so we excluded these categories.

| Snapshot Type | Group 1 >= 9 | Group 2 < 9 |
|---|---|---|
| Infrastructure | 34% | 39% |
| Advanced infrastructure | 11% | 19% |
| Functional | 28% | 6% |
| Example | 4% | 3% |
| Complete | 23% | 14% |
| Lateral | 24% | 10% |

Table 5.3: Proportion of snapshot category relative to total number of snapshots

*The group that produced more snapshots instigated more lateral discussion and produced a greater proportion of functional insight.*

The complete snapshot represents the effect that explanation can have on the audience. Two reasonable avenues for continued research include the study of participant intent to produce complete snapshots and the effect complete snapshots have on an audience.

## 5.3 Summary of Snapshot Theory

Based on data we collected in our study, our critical finding is the definition of the snapshot, which led us to generate Snapshot Theory. Our aim was to understand how a snapshot occurs, what causal and contextual conditions lead to the generation of a snapshot and what categories of snapshots are useful in the organization of knowledge.

Earlier in this chapter, we described the basic categories of snapshots. Here we summarize the main findings that pertain to what we call our Snapshot Theory. The theory proposes that in general, when explaining software, people will use a series of snapshots with particular characteristics and relationships.

The snapshot marks a moment of insight where a component of the meaning of a software model is evident. We now realize that this moment of insight cannot occur until a participant reveals a sufficient amount of information.

The participant must prioritize and filter this information according to the needs of their audience – in this case, a new employee who has recently joined their team. When presented as part of software explanation, a snapshot should not overwhelm the audience's perception process and reasoning capacities, nor should the snapshot ignore the audience's prior knowledge and experience.

Our data suggests the existence of local and global elements of knowledge. Local elements refer to the details of a specific snapshot, and global elements refer to the details that arise from a combination of snapshots. In our transcribed interviews, we coded instances where the participant described many local elements of knowledge in the form of software structure or functionality. We also coded instances where the participant illustrated these local elements of knowledge diagrammatically using a whiteboard. The local elements were bound together into distinct categories, which comprised infrastructure, advanced infrastructure and functionality, that is, how the system appears and how the system works internally/externally. The categories were linked in a linear or non-linear fashion in order to produce global comprehension of a software model, which, in this study, we refer to as a complete snapshot. The theory of snapshots we propose will describe the flow and complexity of a complete snapshot. The complete snapshot is formed as a result of prior snapshots working in unison.

As described earlier, the first snapshot that contributes to the complete snapshot is an infrastructure snapshot. It is a particularly important result of our research that an infrastructure snapshot is always produced. To produce the infrastructure snapshot, the participant recalls details such as the names and functions of the main software components and how the components are connected. The participant then represents the software components diagrammatically using primarily boxes with some text. Typically, the participant devotes substantial time to the annotation of each diagrammatic element.

The next snapshot that generally contributes to the complete snapshot is the advanced infrastructure snapshot. To produce the advanced infrastructure snapshot, the participant imparts further knowledge through a more detailed explanation of one diagrammatic element or software component in the infrastructural composition. The participant provides a thorough explanation of the software component, enriches the original model with more detail (with regular reference to the original model), and often concludes with an evaluation of the model's complexity or state.

With structural information in place, the construction of the complete snapshot may turn to practicality, i.e. how the software functions. The participant reinforces an explanation through the creation of a lateral snapshot that, by adding detail of software

functionality, appears to enrich the listener's knowledge of the fundamental structure. To produce the lateral functional snapshot, the participant adds textual elements to the whiteboard and provides the functional meaning of roughly half of the textual elements. The other half remain unexplained. The participant selects the important components and describes the component's core functionality with ample diagrammatic elements and supporting meaning. The participant then offers an objective evaluation of the system.

Following the creation of a functional snapshot, the participant may produce an example snapshot. The participant starts the example snapshot by posing a practical question and answers the question through an exclusively functional explanation with a variety of diagrammatic elements added to the software model.

The creation of a complete snapshot is not always smooth. Our data suggests the existence of incomplete insight, which we refer to as a 'weak' snapshot. A weak snapshot indicates the absence of explanatory material; thus, through the study of weak snapshots, we should improve our understanding of the conditions for other snapshot categories. The participant starts the weak snapshot with a self-evaluation such as how much information he or she possesses or lacks with respect to the topic. The participant proceeds with an estimate of what the structure of the model should look like through the addition of diagrammatic elements. The participant prematurely abandons functional discussion in favour of general discussion and self-evaluations, which do not adequately fill the gap. The participant does not provide an overarching explanation of the model's functionality and does not link the snapshot with prior snapshots.

In our study, we found that the condition for any kind of snapshot is a sufficient diagrammatic representation linked with sufficient structural or functional meaning. Our data did not reveal snapshots without drawings. Further studies can explore the dependency between diagrammatic material and snapshots.

The core process entailed in the construction of an explanation is the construction of a complete snapshot. The branches in a snapshot event network (e.g. Figure 4.18) reveal the participant's structural organization of key knowledge units with assigned priority. If the participant assigns priority to too many details or lateral snapshots, it is difficult to produce a complete snapshot; indeed, there are times when the participant is unable to reach a

complete snapshot. This structure may take linear or non-linear shape. In non-linear shapes, the participant produces branches of lateral functional snapshots. The complete snapshot may have the following structural organizations:

- infrastructure complete snapshot (e.g. Figure 4.18, Snapshot 7, which had predominantly structural discussion)
- structural complete snapshot (e.g. Figure 4.18, Snapshot 3, without the lateral snapshot, 2.1)
- L-complete snapshot (structural complete snapshot with lateral snapshot) (e.g. Figure 4.18, Snapshot 3)
- L-developed complete snapshot (L-complete with other complete snapshots or developed snapshots) (e.g. Figure 4.18, Snapshot 6).

In the next chapter, we describe our theory in pattern form for the consumption of practitioners.

# Chapter 6 Cognitive Patterns for Software Comprehension: Temporal Details[25]

As we have discussed in earlier chapters, experienced software practitioners use cognitive procedures that have much in common with each other. We have chosen to call these procedures *cognitive patterns*. We discovered the cognitive patterns in software explanation through the field research described in the last three chapters. The cognitive patterns we describe are our interpretation of our study in the context of software modeling. Our goal in this chapter is to convey the concepts from our field research in a format that would be familiar to professional software engineers.

This chapter, therefore, provides software engineers with the application of Snapshot Theory in an appropriate context. We had to abstract away many of our findings in order to get to the essence of what is applicable about Snapshot Theory and how we might present our theory to render it useful in practice. In future iterations of our patterns, we may reintroduce the theoretical findings of the prior chapters as practical use dictates.

We propose that the study of and dissemination of these patterns will help tool designers create more useful tools and directly assist developers in working more efficiently and effectively with software. More specifically, the patterns can be applied to help derive features of tools that aid in *understanding* software, whether for purposes of design, or some other type of problem solving. Such features will support such cognitive activities as reasoning about and thinking about software artefacts.

A cognitive pattern is a structured textual description of a solution to a recurring cognitive problem in a specific context. A general class of cognitive problems in software engineering is the understanding of the structure and function of an object. More specific problems include: determining the most important aspects of a class diagram; understanding how a specific change is going to affect the system; or coping with cognitive overload due to the amount of detail present in a model.

Since cognitive patterns are "patterns," they are related to the design patterns well known in software engineering. But whereas a design pattern captures an effective

[25] This chapter is based on [96] Murray, A. and Lethbridge, T. Cognitive Patterns for Software Comprehension: Temporal Details. *Proc.* 12th Pattern Languages of Programs (PLoP 2005), Champaign, USA, 2005.

technique for solving a design problem, a cognitive pattern captures a technique that is partly or wholly mental and that is employed by practitioners when trying to perform a complex task. Our intent is to translate cognitive patterns into software features that will facilitate a user's cognitive abilities: cognitive patterns are therefore more closely related to usability patterns or patterns of user-interface design. An understanding of cognitive patterns helps illuminate the relationship between user and tool.

All patterns "balance the forces" present in the problem and the problem's context. The designer who uses a design pattern will be interested in the balance between efficiency, reliability, maintainability etc. The person who understands a program will be interested in the balance between cognitive load, correctness of understanding, efficiency of problem solving, and other factors.

Cognitive patterns for software comprehension build on an extensive literature that describes high-level strategies for software comprehension. Well-known strategies include Bottom-up [122], Top-Down [25], Opportunistic [80], As-Needed [85], and Integrated Meta-Model [139]. Each of these may be described as a pattern. The many detailed techniques employed when using each of these can also constitute patterns. In addition, there are problems associated with switching between strategies – for instance, the disruption of the user's mental model [44] – and solutions to these problems can also constitute patterns. Designers of tools such as SHriMP [129] implicitly recognize these patterns and support user needs through a variety of strategies that can be embodied in patterns.

As we construct our understanding of software, our understanding is affected by *time.* Mental models and their internal details change over time. In this chapter, we therefore focus on a high-level pattern for software comprehension[26], termed "*Temporal Details,*" and its subpatterns. The *Temporal Details* pattern illustrates the dynamics of time within the user's mind and helps explain why tools should support these manifest dynamics. Several other patterns contribute to the resolution of forces within the *Temporal Details* pattern. We also present these patterns as a pattern language within this chapter. In particular, we derive the *Snapshot* pattern from our Snapshot Theory and use the *Snapshot*

[26] A complete set of our cognitive patterns for software comprehension, including the Baseline Landmark pattern, may be found online at www.cognitivepatterns.org.

as the fundamental building block of *Temporal Details*. *Temporal Details* are comprised of a composition of *Snapshots*.

## 6.1 Target audience and actors

The instantiation of the patterns in this chapter will contribute to the understanding of artefacts through models. The patterns may seem intuitive and useful to any practitioner involved in software development (i.e. designers, architects, software archaeologists), but our target audience is software developers involved in the maintenance of large-scale legacy software, or tool developers who build tools to aid such maintenance. By using a tool based on these patterns, a software practitioner may be able to locate undocumented design decisions and, as a result, understand the system in its current form by understanding how the system evolved over time. A side effect of this benefit is less reliance on software archaeologists or software gurus.

We will reference three actors throughout the chapter:

- The tool designer, that is, the designer of a modelling tool;
- The model builder, or modeller, who constructs the models; and
- The tool user, designated by 'you' or 'the user,' who uses features based on the instantiation of the patterns when working with models.

## 6.2 Case studies woven through the chapter

We developed the patterns we present in this chapter through three research techniques: a study performed in an industrial setting, the cross-referencing of literature and other field studies in which we have participated. Many of the terms we use in this chapter (including "manipulate", "meaning", "pattern", etc.) can be found in the glossary. To derive the "known uses" in the patterns below, we studied field experts as they worked with several tools, including:

**Grep**: Grep is one of the basic tools used by software engineers to learn about source code. Researchers at the KBRE lab at the University of Ottawa observed software engineers making extensive use of grep as they tried to understand a system component [78]. Some word relevant to the problem would come to mind, and they would issue a grep

command to find all the lines of code in the system containing that word. They would then *store* that grep result. This process would be repeated many times, so that in the end they would have effectively built a *model* consisting of numerous search results in files. They would frequently refer to the stored results, perhaps searching inside one set of results to create another.

**TkSee**: The KBRE group at the University of Ottawa developed TkSee [63] to assist software engineers in program understanding activities. In some sense, TkSee is a 'visual grep' tool: it helps people build, manage and manipulate models consisting of search results.

**IBM Rational Software Architect (RSA)**: RSA enables large-scale team development: many people with different perspectives may work within the same context and the same artefact base and build different views, which may then be synthesized or rationalized into a consistent whole. We examined numerous features, including browse-diagram (a temporary, non-editable diagram that provides a quick way to explore existing relationships among elements), CVS Annotate (a feature within a Configuration Management (CM) System[27]), and Compare-Merge (another CM feature for teams to compare and merge software models).

**Whiteboard Think-aloud Study with Mitel and IBM**: As described in Chapters 3 and 4, we asked software engineers explicitly to explain the architecture of a system they were developing. Many of the patterns we describe in this chapter have been developed based on these studies. We were particularly interested in the sequence of behavioural states that prompted participants to describe complex software architectures in a particular way.

**Chess system**: We developed a tool [133] to allow chess grandmasters to analyse various chess scenarios (games or game fragments known as 'variations'). Grandmasters analyse chess variations while they are playing (and therefore, entirely in their minds), but we are primarily concerned with the physical consequence of their analysis using tools *after* their games. They may perform analysis for a variety of reasons: to improve their

[27] RSA supports Rational ClearCase and CVS; however, in our study, we evaluated CVS.

play, to find 'chess truth,' or for publication purposes. We use this as an example of how many of the patterns apply in a domain other than software comprehension.

## 6.3 Temporal Details

In this section we first describe the *Temporal Details* high-level pattern, and then discuss several other patterns that relate to *Temporal Details*.

Figure 6.1 is a pattern schematic that illustrates the related patterns. The patterns shown in Figure 6.1 combine to resolve the forces of *Temporal Details*. The circles represent the *Snapshot* pattern (important moments in time). *Snapshots* contain *Meaning*. The boxes represent groupings of *Snapshots*. The arrows represent transitions between *Snapshots*.

There are several kinds of groupings of *Snapshots*, each shown as a box. This figure shows: a *Longview* (sequential *Snapshots* that tell a story), *Multiple Approaches* (parallel *Snapshots*), a *Snapshot* chosen from a set of *Quick Starts* (the evolutionary origin), and finally a sample of the ability to *Manipulate History* (which illustrates how we manage evolutionary complexity).

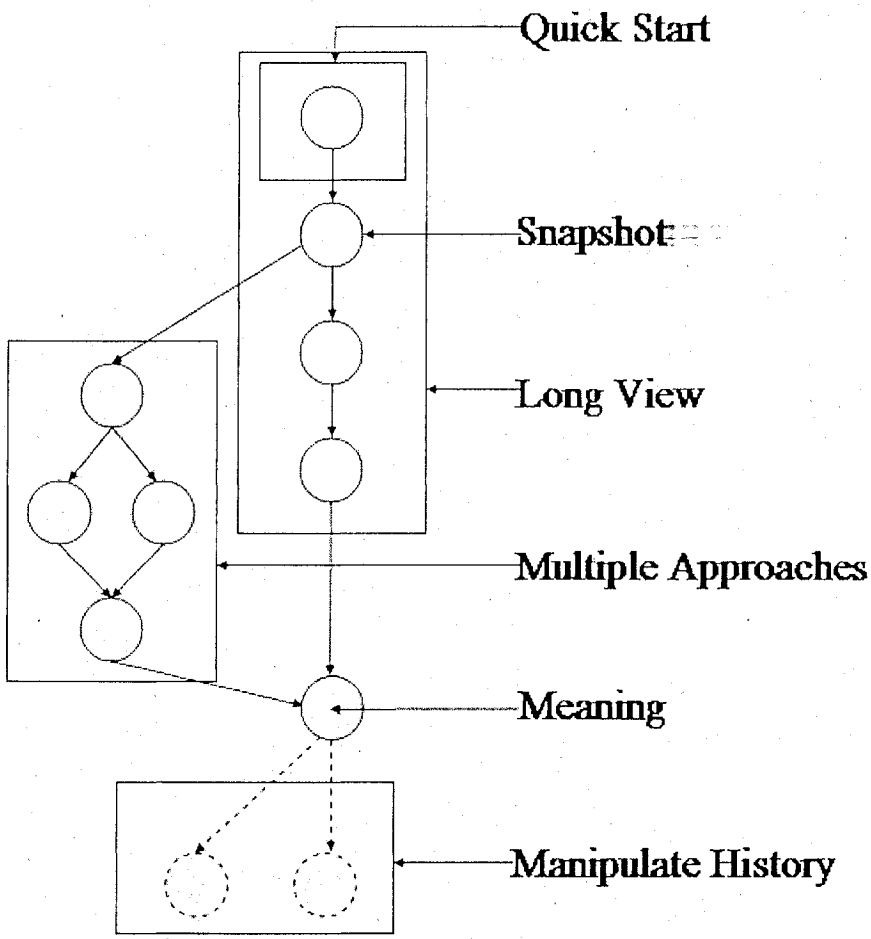


Figure 6.1: *Temporal Details* Schematic

## *Temporal Details*

Context: *You are dealing with models that evolve over a period of time.* You may be using a tool to explore, reverse engineer, document or explain a large-scale system (e.g. software). Alternatively, you may be doing these activities by drawing diagrams informally on a whiteboard. The models can be diagrammatic or textual.

Problem: After editing a model, you expect that the final version will maximize understanding of some aspect of the system. However *the final model lacks intrinsic details that enable certain kinds of understanding.*

Forces:

- *A final model may be difficult to understand due to its size and complexity.* To understand the way a system works, it may be better to look back to a time when

life was simpler. For example, attempts to make sense of the physical world using our current understanding of the laws of relativity are very difficult. If instead you base your initial understanding on Newton's laws, you will learn much faster, even though Newton's laws are not entirely correct for large speeds and masses. Newton's laws provide a simpler view of reality.

- *To understand a system you may need to understand it incrementally.*
- *To work with a system concretely you need the final model*, since earlier models will be incomplete and inaccurate. Earlier models have inaccuracies or gaps preventing you from being able to properly understand, develop or find the flaws in the system.
- *The final model contains many levels of detail but abstracts away key historical decisions and their rationale.* A tool supporting a block diagram of a CPU allows you to choose your desired level of detail by drilling down and backing out of different levels, such as sub-blocks, circuits, and transistors; this type of detail we call 'drill-down details'. Drill-down details are needed to tell you how a system works and allow you to use a system concretely, but *they do not help you understand why the system is the way it is today.* Drill-down details do not support an understanding of the earlier decisions that constrain the current system.
- We rarely have the luxury of building a system afresh; therefore, *we need to understand and cope with the present constraints*, which are there due to historical decisions and prior constraints. If we built the system afresh today, the constraints may be different, leading to a different design.
- A prior model of a system may be unsound, or incorrect by the standard of today's system, but *understanding prior models is needed to make effective decisions and to avoid making recurring mistakes.*
- *The historical information may be too rich and complex and thus confound the person working with the model.*

Solution: The tool developer must support the ability for tool users to view and manipulate aspects of history and the decision-making process that went into the development of the final model. The user will then be able to more easily understand why certain decisions were made early in the model's development, and will be able to

understand the present system as an extension of the simpler system that once existed. The user will always have a choice: to simply view the final form of the model, or to explore the dynamic details of the model's creation. Such details could include the different versions of the model over time, key decisions made and their rationale, and thought processes applied as the model was evolving. Ideally, a tool that supports such details would be transparent, so that users who do not need to look back at this history would never have to contend with such a feature. Those users who need historical information can view whatever levels of history they choose.

Known uses: There are a variety of software engineering tools designed to capture design rationale [66, 73, 91]; these originated in the user-interface design community and enable developers to store information such as the decision tree leading to the final model, as well as the logic of each decision. Configuration management tools explicitly store states of a system at certain discrete points, and encourage annotation of the changes made each time a version of the system is saved. As an example, corporate meeting rooms often have equipment to create a digital image of the contents of a whiteboard. Also, learning the historical refactorings helps a developer understand the present state of a system. The *Temporal Details* pattern describes how one can capture and illustrate rationale and history in a broader sense than any one of the above tools.

Resulting Context: The application of *Temporal Details* and its related patterns will result in *tools for explaining, exploring and documenting systems* that take advantage of the knowledge embedded in a model's history and *that mesh more closely with the way users think and act.* Full application of *Temporal Details* would result, for example in the ability to *Manipulate History.*

Important downsides of the pattern are that the environment must be built and the historical data must be managed on an ongoing basis. In other words, a tool for working with *Temporal Details* must leave a footprint that is greater than would otherwise be left by a design tool.

Issues that arise when applying *Temporal Details* include deciding which historical information to capture, how to represent it, and how to manage details that accumulate over time. Some of these issues are addressed through the modeling of software evolution by

treating history as a first class entity [43]. If these issues are managed well, each user should be able to choose the level of understanding that is appropriate to that user. At the root of the resolution of these issues is the following question: What are the key historical moments, the moments that exemplify key insight? The challenge posed by this question is addressed in the *Snapshot* pattern.

### *Snapshot*

Also Known As: Short View, Temporal State, Coherent Point, Conceptual Whole, Cohesive Nugget

Context: You are *working with a single evolving model* in the context of Temporal Details. Any model is modified by a series of operations, such as adding or removing model elements. The elements could be as small as single lines or characters, or could be larger compositions.

Problem: *With what granularity should you track the evolution* of the model? That is, what are the most useful points to stop and think about the model as it is being built, or look back later to see how the evolution occurred?

Forces:

- *A concept*, even in the context of a complex model, *can often be conveyed simply.*
- *Small but self-contained changes to a model may convey important new meaning.*
- Building a large, complex model before stopping to think about it is likely to result in confusion, and the skipping over of important learning.
- Adding, deleting or replacing information in a model can convey new understanding.
- Tracking evolution with an excessively coarse granularity fails because *humans need smaller increments to evolve their understanding*, and you lose historical perspective.
- *If every gesture* involved in evolving a model *is tracked*, then we are guaranteed that *no historical information is lost.* But to step through each possible state of the model *would be slow*, and we would lack guidance about which points are salient.

- Some states (e.g. after a single line is drawn) are incoherent or *represent incomplete concepts*. This level of granularity is too fine.
- *When humans try to learn in large numbers of very fine increments, they may not understand the big picture and retain the details.*
- *A model does not have to be complete and accurate before one can pause to think.* Neither completeness nor accuracy is needed for incremental understanding; striving for these will be important for some purposes, but not for the gain of understanding.

Solution: *Track and capture evolution at moments when the model is cohesive or conceptually whole.* We call these moments *Snapshots*. Our research shows that *Snapshots* are naturally present in the development of a model.

By cohesive and conceptually whole, we mean that a concept being conveyed in the model has had sufficient detail added so a person studying the model can understand the concept, but not necessarily perfectly. The granularity of tracking will therefore depend on the strategies used by the person doing the modelling (the *modeller*): Some modellers might add seemingly unrelated elements to a model, and only after considerable time, link them in a way such that the model becomes cohesive. In this case, the *Snapshots* will be far apart in time. Others modellers might add very small increments such that the model is highly cohesive after each increment. In either case, being able to view the model as it existed at *Snapshots* will group potentially large sets of model states into more manageable units.

The *Snapshot* is a coherent step in the process of evolving a model towards its final form. The *Snapshot*, irrespective of the underlying meaning of the model it conveys, will reveal nascent information either independently or in conjunction with other *Snapshots*. It is not generally worth spending a lot of time designating *Snapshots* with a very high level of accuracy; a rough approximation of the set of *Snapshots* will often be sufficient to achieve the objectives of this pattern.

In a tool, a *Snapshot* might be created manually through user actions including 'tagging', saving, annotation, etc. A *Snapshot* might also be identifiable automatically: A modelling tool might recognize pauses; i.e. the tool might automatically tag a version as a

*Snapshot* when the modeller has made a collection of changes, and then pauses before making more changes. If it is a graphical modelling tool, the tool might recognize the completion of certain diagrammatic 'idioms' (e.g. drawing two boxes with a line between then and labelling the boxes), or even capture the oral explanation made about a diagram. We have used these approaches when manually determining the *Snapshots* in a model's history.

Resulting Context: The application of the *Snapshot* pattern to a tool will enable users to build and present a model in appropriately sized increments, and allow the user to reference and come back to some of those model versions if needed.

How to identify a *Snapshot* is both an empirical and implementation challenge; what constitutes a conceptually whole moment will be subjective. A tool for working with *Snapshots* will need to allow for this subjectivity and imprecision. Also, the *Snapshot* pattern does not suggest how to organize the entire set of *Snapshots* over time, nor does it suggest how the *Snapshots* ought to be presented. We need *Long Views* and *Multiple Approaches* to organize *Snapshots*, and we need a way to *Manipulate History* for further organization and presentation. When we want to build or recognize the first *Snapshot* of a new model we may need to choose a starting point from a set of *Quick Starts*.

Another problem with *Snapshots* is that although state alone may convey some meaning, the rationale or the decisions made to arrive at a *Snapshot* are not always conveyed in the model – further explicit *Meaning* associated with *Snapshots* may be required. Further to this point, a *Snapshot* does not give you the whole picture; you only get the picture at a moment of time.

Related patterns: The *"Speculate about Design"* pattern [39] calls for a software engineer engaged in software reengineering to refine their model of a system by checking hypotheses about a design against the source code. In this case, a *Snapshot* can be constructed for each hypothesis. To *Speculate about Design*, an engineer inserts open questions as notes into a software model, then iteratively addresses each question and refines the model accordingly. The reengineering process builds a series of *Snapshots*.

The "*Migrate Systems Incrementally*" pattern [39] encourages developers to avoid the complexity and risk of "big-bang reengineering" by deploying incremental updates of the system. In this case, each update can be considered a *Snapshot*.

The "*Just Enough*" pattern [87] tries to ease learners into the more difficult concepts of a new idea by the provision of a brief introduction and dissemination of more information available when the learner is ready for it. In other words, *Just Enough* describes the division of information into coherent units and a way of delivering information from the learner's point of view. *Snapshots* similarly tackle the "right" amount of information and the user's understanding. However, a *Snapshot* is a state in an ongoing process leading towards creating a final model: The emphasis in a *Snapshot* is keeping a point in the history of the model in case it may be useful, whereas the emphasis of *Just Enough* is the active design of an increment in a learning medium. Also, the *Meaning* behind the *Snapshot* is considered separately.

The "*Step by Step*" pattern [87] encourages people to tackle problems in small increments "with short-term goals, while keeping your long-term vision". Incrementality is therefore a common feature of both *Step by Step* and *Snapshot*; however, in *Snapshot* the idea is to review prior increments, rather than to work forward in increments.

Known Uses: Many tools offer an ability to examine a specific model at a given moment in time, in other words to take a *Snapshot*. This *Snapshot* is not always what the user is looking for in terms of the particular details, but aids understanding.

Grep: Grep produces a model of some aspect of the static state of a system based on a specific set of queries – the effectiveness of the results is directly proportional to the effectiveness of the queries, but rarely will an individual query generate everything the designer needs to solve a specific problem, and never will a grep query represent everything in a design. Nevertheless, individual grep results can be extremely useful *Snapshots* in the user's evolving understanding of a problem. In our studies we observed software engineers print out grep results, store them in files and use them as checklists.

TkSee: TkSee was specifically designed to enable people to explore software and incrementally build models. The models are 'history states' that show certain patterns of

relationships that bear on the current problem. As with grep results, software engineers discard these after a short period of time (several hours to several days).

RSA: RSA supports *Snapshots* with the Compare-Merge feature. A *Snapshot* in this context refers to the individual differences between two versions of a software system. If one of the versions is the present system, and the other version is a point in local history, then each *Snapshot* represents an evolutionary development. Figure 6.2 illustrates a tree structure for navigating the differences, and Figure 6.3 illustrates the differences visually (e.g. in this particular case, a circle shows the dependency between Class1 and Interface1 has been removed). In addition, CVS versions are *Snapshots* of an entire system in a moment of time, although often with a granularity that is much higher than what we envision for a *Temporal Details* tool. RSA also supports individual browse-diagrams for analysis; these can be seen as *Snapshots* supporting partial visualization of a system in a moment of time.

Studies at Mitel & IBM: In many instances during our videotaped analysis, the participants would produce diagrams on the whiteboard and then begin discussing them. They would not speak until they had made enough changes so the diagram was in a new coherent state: they were thus building *Snapshots*. Based on the questions asked, the participants would then iteratively produce refined *Snapshots* of the system.

Chess system: In chess, a *Snapshot* corresponds to a particular board position. In the mind of the grandmaster, he is examining not only the *Snapshots* that exist on the board, but also various interrelated moves that may occur (*Snapshots* from variations) [72]. A tool to support analysis must support not just the main board positions, but also variations.

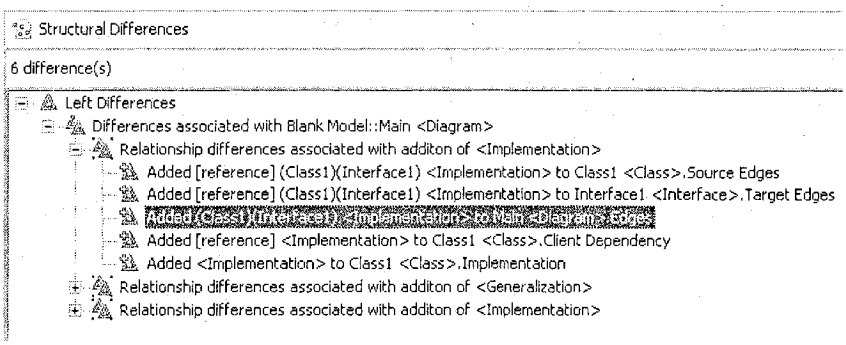


Figure 6.2: Tree-based navigation of Snapshots in RSA

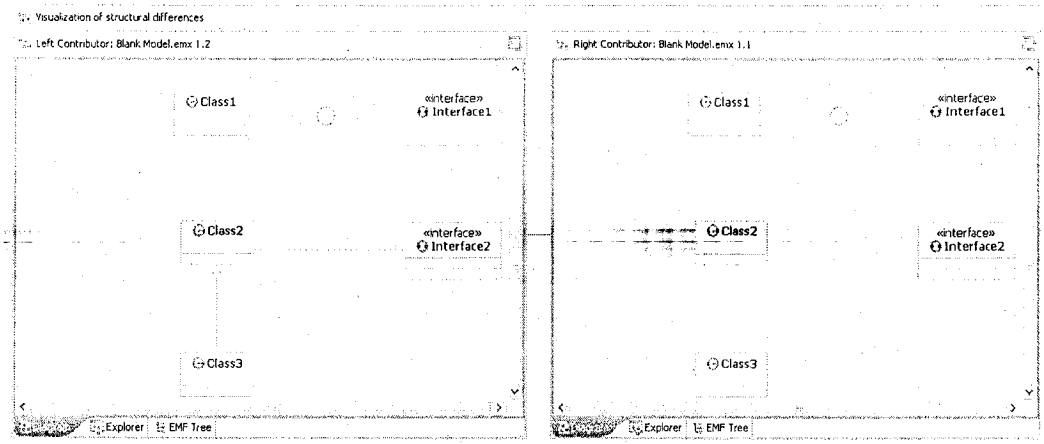


Figure 6.3: Visual illustration of *Snapshots* in RSA

## *Quick Start*

Also Known As: Starting Templates, Library of Openings

Context: You are creating a model in the context of *Temporal Details*. This might mean you are starting from scratch, or you might have already evolved a model through many *Snapshots*.

Problem: How do you effectively and efficiently create new models?

Forces:

- *A model will always have at least one diagram (and hence one starting point), but will often have many.*
- The start of any task is often undertaken instinctively. However, *people often have difficulty when they start a task.* On the one hand, they may need a catalyst, and on the other hand, they may strive for an immediate, though unobtainable, perfection.
- People may start with anything at all, just to get themselves going. Consider, for example, how the US Marines start with something called a "70% solution" [51] that encourages "high tempo." The idea is that it is better to decide quickly on an imperfect plan than to deliver a perfect plan too late. The Marines find the essence of a complex situation and build upon it quickly. Professional writers also use this technique.
- *People need quick ways to plunge into a new task*, such as modeling, which are not inhibited by start-up costs. These costs might be the need to determine where to

save something, the need to make links to existing models, or the need to construct a well-known canonical starting point.

- *If people do not have a familiar place from which to start, understanding is inhibited.* But then, if they start with something too large or irrelevant, significant time may be wasted adapting it. People need a familiar or central starting place of an appropriate level of complexity when they begin a process of understanding.
- *An explanation is often best when it contains many interrelated models*, each of which has to be created, and therefore must be started at some point in time.

Solution: *Allow users to quickly start new models by choosing from a small set of existing simple models*. The selection of starting points is more appropriate if they represent a familiar landscape to the user. Each starting point is a *Snapshot* of a model that will likely then evolve through many more *Snapshots*. This first *Snapshot* is neither trivially small, nor is it too complex. As with all *Snapshots*, it is a view that is coherent enough to be talked about.

In a tool, this pattern can be implemented using templates of sophisticated – but still very simple – designs. For example, it might be useful to start a state diagram with two states linked by a transition: not many useful state diagrams would ever have fewer states than that.

Resulting Context: This pattern encourages speed of exploration or explanation, as well as the creation of multiple models. After you have chosen one of the *Quick Starts*, one of the hardest decisions – where to start – is behind you, at least until you choose to start afresh. A *Quick Start* provides a guide as to where to go next, and how to keep going. As you build a series of new *Snapshots* based on your *Quick Start*, you may need to consider if you intend to build in sequence (*Longview*) or in parallel (*Multiple Approaches*).

In creating a list of suitable *Quick Starts*, you must determine which ones will be appropriate. A downside of starting with simple and imperfect starting points is that you may also need to start again several times. However, this may support incremental improvement of your understanding of a complex system. The real danger comes from the reliance on your set of starting points as definitive, that is, a reluctance to explore other possibilities.

*Forcing* people to choose a particular starting point would be contrary to the pattern. Users often find they stick exclusively to the templates provided or spend too much time exploring the template set.

Known Uses:

Studies at Mitel & IBM: Our participants start with a single notion, usually a *Baseline Landmark*, and incrementally expand from this starting space. In the telecommunications domain, for example, it is very common to draw a diagram of a "plain old telephone system" (POTS) and explain some new feature by evolving this diagram.

RSA: A software architect may either create new model elements or access existing assets (e.g. requirements code, other models) to build a model. An architect may use search and navigate features to access existing features (*Baseline Landmark).*

Chess system: In chess, grandmasters use "critical positions" to study opening theory. The critical position may be a hotly contested position – perhaps many other professionals reach this position in their play often, or the position is deemed "OK" since a first-class grandmaster played it recently. The grandmaster may prepare for a future opponent by analysing *Snapshots* from the opponent's opening repertoire. Or the grandmaster may wish to broaden his repertoire. In all cases, the grandmaster starts analysis from a historical and relevant position.

Other tools: Word processors and presentation software provide libraries of templates to allow *Quick Starts.*

## *Long View*

Also Known As: Highlight the Story, Higher-Order Snapshot

Context: You are evolving a model through a set of *Snapshots.*

Problem: *How can you tell an effective story* when the complete set of *Snapshots* comprises a rich set of details?

Forces:

- *People appreciate a story* – a tale that evolves over time where linkages are made from step to step.

- *If there is no connection* between historical steps, *then there is no story.*
- An individual *Snapshot* conveys some *concepts*, whereas others *emerge only through a sequence of Snapshots.*
- *One can lose sight of the forest* (a story) *for the trees* (the individual *Snapshots*).
- *If all historical steps are connected, it is hard to see the key steps.*
- *People have difficulty comprehending a 'big bang' explanation.*
- Explaining or exploring based purely on a series of unrelated *Snapshots* fails because the *sequencing of Snapshots helps incrementally build understanding.*

Solution: Allow users to look at the history of model evolution and designate a coherent sequence of *Snapshots* as having particular significance. The sequence tells a story that would otherwise be hidden. The end-points of the sequence become, in some sense, higher-order *Snapshots*. A *Long View* is a view that shows how something evolved. How something evolved from one *Snapshot* to another is a story. Sometimes you cannot grasp a concept unless you have more of the small units. Sometimes you cannot grasp one of the small units unless you grasp another one. By seeing the individual *Snapshots* in context you may be able to understand a larger component of the entire system or understand some otherwise unintelligible *Snapshot*.

Resulting Context: You may start a new *Long View* to explore a new aspect of the system; you may also start a new *Long View* if another was less fruitful than expected or resulted in only a partial understanding. A tool that implements the *Long View* pattern would allow the explanation and exploration of the history of a model through sequences, rather than just a simple presentation of *Snapshots* without any organization. Someone understanding a design would be able to understand more about the thought processes of designers. However, just as all *Snapshots* need not be retained, the user needs to remain flexible as to which sequences will be stored for later reference – and needs to remain in control of those choices. Tools to help guide the user through a series of *Snapshots* may also use "relevance feedback." You may still need a mechanism to *Manipulate History* to clean up and organize *Long Views,* as well as to add *Meaning* to them.

Known Uses:

Grep: We have observed software engineers doing two things to evolve and group their query results: one is to edit their previous grep command lines; the other is to pipe one set of grep results through another grep query. The notion of revisiting previous queries is poorly supported by tools. With native UNIX environments the user does however have access to buffered history of terminal output, and the history of commands. Users can also save queries to text files that can be concatenated, or run through other grep filtering steps. Taken together, these facilities allow a rudimentary ability to create a story or *Long View* from query results.

TkSee: TkSee contains multiple indented trees of queries. The user can easily refer to previous queries, and group several of them together as a *Long View*. Users can also save query result sets to files and bring them back.

RSA: The Compare-Merge feature supports a series of *Snapshots*, which collectively form a *Long View*. If the software architect clicks on consecutive items in the tree shown in Figure 6.2, RSA illustrates the *Long View* visually, as in Figure 6.3. RSA compiles a list of versions in the "CVS Resource History" which illustrates the *Long View* as an evolution of a system across versions.

Studies at Mitel & IBM: In our whiteboard video sessions, the participants would iteratively refine diagrams representing their knowledge of a software system. Because of limited whiteboard space, they would erase (cull) less important aspects of the model to provide more relevant artefacts to address the questions. History was difficult; to refer to previous versions they would use verbal comments or redraw previous diagram components.

Chess system: While analysing a position, a grandmaster first compiles a list of candidate moves, the *Quick Starts* for branches of forced (through a series of checks or threats) and unforced variations. The grandmaster subsequently steps through each candidate move in turn, building a *Long View* for each candidate move. The *Long View* is a series of moves that ends in an evaluation (white is winning, the position is equal, the position is unclear, etc.). For many positions within the variations, the grandmaster must

compile a new set of candidate moves; thus, the grandmaster is building a tree of variations – the topic of the *Multiple Approaches* pattern.

### *Multiple Approaches*

Context: You will be modeling or understanding using a set of *Snapshots*

Problem: How do you support the representation of non-linear patterns of evolution of a model?

Forces:

- *People* may not fully understand one explanation, and *may need an alternative perspective.*
- *Different people may need to approach understanding using different strategies.*
- *There may be different* but perfectly *valid ways to model something* or solve a problem.
- Allowing only a single path fails because it does not recognize the alternative perspectives, or else forces the perspectives to be considered in a less useful order.

Solution: *Allow branching and re-joining in the network of connected Snapshots.* After a branch point, the *Snapshots* or sequences of *Snapshots* (*Long Views*) in either branch can be used for different purposes. A user may use different sequences to explain or explore the various aspects of a system. Alternatively, the user may use multiple sequences to approach the same aspect in different ways, either developing an alternative understanding, or a more complete understanding. Sequences may split and merge at arbitrary points.

Resulting Context: A user will be able to designate and explore multiple paths for understanding and exploration and the user should see the path structure so he or she can compare paths and learn from different perspectives. One obvious downside to *Multiple Approaches* is the generation of a complex network of model versions. The user may need to *Manipulate History* to cope with the large amount of information and to filter important details.

Related Patterns: The *Multiple Approaches* pattern allows a person in an organization to make a compelling argument from different viewpoints as to how an idea may meet the

*Tailor Made* [87] needs of an organization and the *Personal Touch* [87] that people require to see the personal value that an idea may bring them.

The *Multiple Approaches* pattern depends upon branching. A group of patterns which addresses branching from the perspective of software configuration management (SCM) is "*Streamed Lines*: Branching Patterns for Parallel Software Development" [12]. The SCM patterns describe how to support parallel development through project management, organizational structures and technology. The implementation of the SCM patterns helps address problems of communication, visibility, project planning, and risk management and resolve some of the technical challenges associated with the capture and organization of *Snapshot* networks.

Known Uses:

TkSee: We studied users and evolved TkSee with features to build and explore hierarchies of exploration paths incrementally: each branch is a separate approach, throw away exploration paths or sub trees, save exploration trees to a file and reload them and, switch among trees. In fact, TkSee supports a hierarchy of exploration hierarchies.

RSA: A software architect may analyse *Multiple Approaches* with browse-diagrams (system *Snapshots)* with the intent of building deeper understanding through multiple perspectives. RSA is not limited to browse diagrams – a software architect may create many typical UML diagrams (*Snapshots*) and the tool allows the architect to link them all. Thus the tool supports multiple branches and joins between *Snapshots*. In addition, RSA supports *Multiple Approaches* through CVS features such as branching (that is, you retain the baseline while you work on different versions).

Studies at Mitel & IBM: In our whiteboard sessions, the participants digressed into discussion of seemingly unrelated parts of the system, and later conjoined concepts to deepen understanding.

Chess system: As has been discussed, during analysis grandmasters built a tree of variations for candidate moves, or *Snapshots*. The right pane in Figure 6.4 illustrates support for *Multiple Approaches* through a visual hierarchy of clickable moves. When a grandmaster clicks on a move, the system illustrates the *Snapshot* in the left pane.

Other examples: Multiple product lines, multiple configurations, achieving the same result by applying different methods.

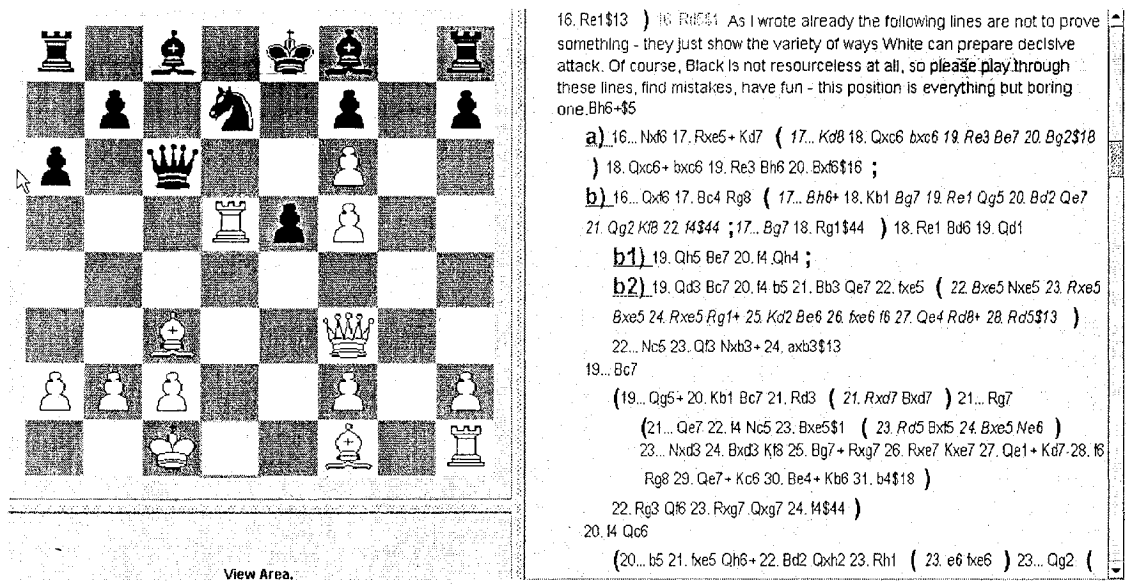

Figure 6.4: ChessLink Analysis Feature, *Multiple Approaches* as chess variations

## *Manipulate History*

Also Known As: Superman rewinds time to save Lois Lane

Context: You have a historical record of model evolution in the context of *Temporal Details*

Problem: The network of *Snapshots* may not be good enough for users to learn from.

Forces:

- A, then B is the way things happened, but B then A may make a more comprehensible story.
- *History is in the eye of the historian and the reader of a history*: No two historians will tell a story the same way, and no historian will ever know exactly what happened.
- Historical fiction can help one understand history by making it more comprehensible. No harm is done as long as one realizes that it is not a literal representation of past events.

Solution: *Allow for the network itself to be rearranged and adjusted* so you can revisit your understanding process by following previously followed paths. But as you do this, retain the previous network. The result can then become a network of networks.

Resulting Context: You will be able to edit not just the models, but also the networks of models. This pattern builds on *Snapshot*, *Long View* and *Multiple Approaches*: Those patterns allow you to designate points, sequences and branches in the history of a model's evolution. *Manipulate History* allows you to adjust that history itself.

Known Uses:

TkSee: Supports the manipulation of multiple hierarchies of historical queries. For example: *Snapshots* are the results of queries or other operations on the model; *Long Views* are the sequences of queries that form a hierarchy whose results can be saved as an exploration and revisited later; these explorations can themselves be edited to refine the user's understanding, and saved again.

RSA: Using the Compare-Merge feature, developers can retrace the meaning of decisions by interpreting iterative changes. Furthermore, UML notes, comments and documentation attributes may provide insight into the decision process.

Chess system: After a chess game, grandmasters investigate what-if scenarios for the critical moments of games to unlock the 'secrets of the position'. The primary goal of this is to improve their thinking, though they may also uncover improvements in their games that they can use in later games. The entire basis of modern chess opening theory is continual reflection on revision of historical games [72].

### *Meaning*

Also Known As: Annotation, Metadata

Context: You are working with a network of *Snapshots*.

Problem: A model cannot inform you why the decision to transition from *Snapshot* to *Snapshot* was made, yet you often need such deeper understanding

Forces:

- *A deeper understanding of history can be derived if you know why something* happened, not just what happened. The "why" is, however, often lost in the mist of time.
- A *Snapshot* only captures state and will often not even imply the rationale for the state.
- Tying rationale for a *Snapshot* to the *Snapshot* itself may be inappropriate as it is may be difference between two *Snapshots* that is of most interest – and one may later on want to *Manipulate History*, which would seriously confuse rationale tied to a single *Snapshot*.
- Rationale attached to a *Quick Start* may facilitate its use.
- The need for annotation is proportional to the size of a network of historical models.
- People often want to make use of information about information (meta-information) that may be valuable.

Solution: *Allow annotation or other mechanisms for recording knowledge about any of the Snapshots or transitions between Snapshots.* Annotation features in tools may be one step towards retaining *Meaning*. Clearly developers need to indicate significant information that cannot be represented in diagram form. Perhaps notations need to be designed to represent this kind of information; or at the very least, structured documentation formats could be developed to capture this information.

Resulting Context: Following use of the *Meaning* pattern, the tool designer can build tools to allow annotations of all types. For example a tool that stores several states of an evolving explanation could allow the user to record why new details are added or replaced to create a new *Snapshot*. Design rationale is an important area of study in software

engineering. Tools should allow the user to flow more easily from one design task to another while storing design decisions. Downsides to this pattern occur because people often disdain documentation. More annotation implies three more work tasks, one task is the annotation step, the second is the maintenance of previous annotations, and the third is reading annotations. Transparency is a very important aspect of this feature: do not enforce any of the three prior works tasks, but support them to an appropriate degree. Simply marking *Snapshot* moments may be sufficient *Meaning*.

Known Uses:

RSA: The developer may commit changes to CVS with comments annotating their rationale (illustrated in Figure 6.5). Furthermore, a developer may use the "CVS Annotate" feature (illustrated in Figure 6.6) to associate changes and comments with a particular version. In addition, RSA has a traceability feature supporting traceable design decisions across artefacts.

Studies at Mitel & IBM: The participants described their rationale for drawing. In other words, they described why they were about to start explaining something; why they were erasing something or why they were adding new details. Our participants provided more of the structural or functional meaning of the model than their rationale for drawing.

Chess system: Grandmasters analyse their own games to determine the "truth" behind the decisions they made over the board. They record these analyses in both variations and textual annotations, particularly when the games are to be published. The right pane in Figure 4 illustrates a sample annotation.

Revisions of '/ModelSystem/Blank Model.emx'

| Revision | Tags | Date | Author | Comment |
|---|---|---|---|---|
| 1.3 | | 5/9/05 4:32 PM | hfarah | added a Utility class |
| 1.2 | | 5/9/05 4:31 PM | hfarah | added generalizations and implementations relations |
| 1.1 | | 5/9/05 4:29 PM | hfarah | initial submission |

Figure 6.5: CVS commenting in RSA, one form of *Meaning*

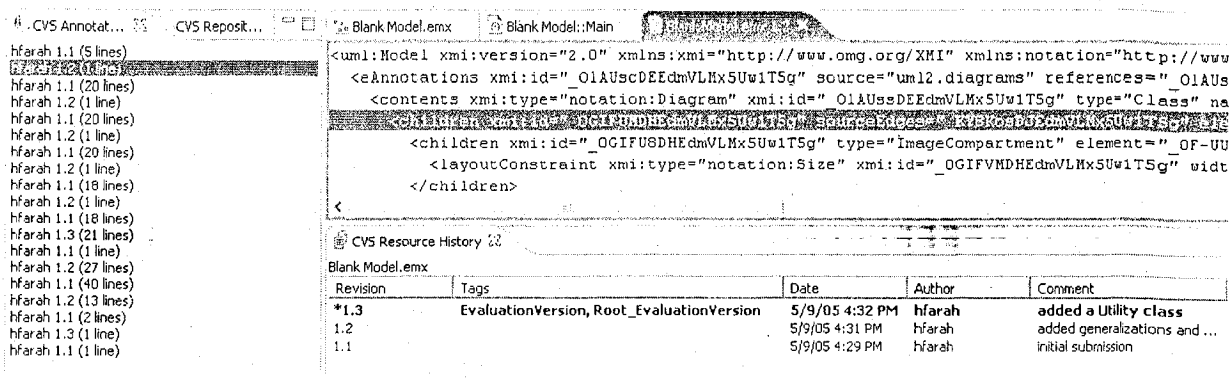


Figure 6.6: CVS Annotate feature for visualizing changes and comments with versions

To sum up then: we can leverage the patterns now expressed to gain a deeper realisation of their benefits in forthcoming tools.

## 6.4 Epilogue

In this chapter, we presented the notion of cognitive patterns, and outlined a pattern language for software comprehension. We focused particularly on the *Temporal Details* pattern language, showing how we derived this language from field research and how it can be applicable to tool builders.

Far more than an attempt to put a new face on certain program comprehension literature, cognitive patterns constitute an attempt to tie many results into a rich, new language. Our objective is that this language should provide perspective, structure and a common lexicon that can be used by tool designers and academics to relate practical problems in software comprehension. However, this places a heavy burden on cognitive pattern authors: what constitutes 'practical'? The authors of this paper consider a pattern practical if it addresses a recurring problem that the community presently cares about, and is genuinely of use to tool designers. We have tried to balance the following criteria: a) ensuring patterns compose well; b) ensuring patterns reflect cognitive issues users encounter; c) grounding them in reality; and d) making them useful to tool designers.

# Chapter 7 Threats to Validity

"Qualitative analyses can be evocative, illuminating, masterful -- and wrong" [93, Qualitative Data Analysis, Miles and Huberman, pp.262]

## 7.1 Introduction

In this chapter, we address threats to validity. If we were to ignore these, it might suggest that we suspected our qualitative research approach or results to be biased, inaccurate or imprecise. Where possible, we will provide evidence showing how we reduced the threats, and where such evidence is not available we will argue why our work is nonetheless legitimate and reasonable.

In general, the emphasis in grounded theory research should be on the *generation* of theory. Some research methodologies have a deep focus on the validation of results; but grounded theory focuses on clarifying bias and assessing how threats to validity affect the methods or results. Wolcott, for example, states that unequivocal determination of the validity of findings is impossible [147]. Erickson states that our aim is to be explicit about our biases, not to persuade anyone of their superior virtue or even their reasonableness [45].

Throughout the execution of our study, we were critical of the representativeness, the reliability and the replication of our findings. In this chapter, we address the rift between cognition and explanation, study limitations (i.e. time, medium, site and interview questions), theoretical and practical coding threats, misrepresentative data, researcher effects, internal validity, external validity and the applicability of our patterns.

In the following sections, we label threats 'T<threat id>' and threat resolutions 'R<threat id>'.

## 7.2 The Explanation-Comprehension Rift

T2.1: The explanation of a software system to others is different from one's personal comprehension of a system. The researchers claim to have studied comprehension when in actual fact they studied explanation.

R2.1: We speak to this threat at length in Section 2.4. There is precedent for studying comprehension through the explanations of professional software engineers [13, 61, 80, 112]; researchers analyse fragments of software explanation to find recurring behavioural patterns that provide clues about internal cognitive events. The basis for this approach is the assumption that the verbal statements of professional programmers provide the closest approximation to a trace on the participants' thought processes [80]. In following this assumption, the prior research does not acknowledge that the goal of software explanation is software understanding. In our research, we structured interview questions to prompt a large number of inquiry moments, which Letovsky identified as producing the richest cognitive responses.

The challenge we faced in our research was to be certain that our task invoked moments of software understanding. The response to this challenge is as follows: our participants perceived and interacted with external stimuli, accessed short-term and long-term memory and used external memory to store and organize explanation information – hence, they indeed engaged in cognitive activities as they responded to our questions. Because our study called for participants to sketch while they explained (c.f. [109]), our study is a contribution consistent in its assumptions with prior research. We also add that because we designed a natural task in a familiar environment, we are more likely to determine realistic behaviour than our predecessors [103].

The bottom line is that our participants constructed snapshots to enable comprehension for their audience. The critical point is that if you use a tool to understand software, the tool should enable comprehension through the creation of snapshots.

## 7.3 Internal Validity

To satisfy internal validity, a researcher must show that results are authentic, make sense and are not affected by the researcher's presence and actions. In this section, we will discuss study threats as they concern limitations of time, site, medium and interview questions.

T3.1: Researchers moved from coding to interpretation too early.

R3.1: If we moved from coding to interpretation too early, we would risk not properly challenging plausible concepts with alternative hypotheses. We used the concept of *saturation*, a typical grounded theory approach, to determine when to move from coding to interpretation. As stated in Section 4.3, we identified the point of saturation as the moment when regularities in the data stabilized and further coding and review of data did not yield new categories or codes. We received 100% agreement with three coders that to understand snapshots at a deeper level required interpretation and that further coding would provide no benefit.

T3.2: The timeliness of results is dependent on expansion of modelling languages.

R3.2: The reader may question the longevity of our results, for example, by questioning if the expansion of modelling languages (e.g. UML) into mainstream software development led to skewed results, results that would be different if a similar study was performed in the future. There are a number of arguments against this reasoning.

First, we observed the use of whiteboards, which are analogous to tools people have used to discuss concepts for centuries. Second, our drawing codes were generic to avoid this issue. Third, the major finding is that developers use snapshots to explain software systems and we could not find evidence to suggest a dependence on diagrammatic techniques. Finally, a wide variety of notations were used by our participants, not just UML; we could not therefore conclude that the participants were influenced significantly by current modeling trends.

T3.3: Researchers sampled only 2 sites with only 24 participants.

R3.3: Sample size for the purpose of generalization is not an issue in grounded theory research. Other researchers, (e.g. [13, 80, 93]) studied only 6 participants, while we involved 12 participants per site. This number was adequate in terms of achieving the saturation point. As to our sampling only 2 sites, we have a number of responses. First, we should note that we eliminated bias that might arise through the study of a single company or a single domain of software. Second, we studied professional software engineers: this makes our study design more significant than research involving students. Third, we drew from the results of our first sample (our coding scheme) as the basis for our analysis of the second sample. Fourth, we performed a variety of comparisons across participants, which

resulted in significant adjustments to our theory. We can further reduce this bias in the future through cross-case comparisons of our two samples and further studies with additional industrial participants.

T3.4: The study is biased towards elite participants.

R3.4: This threat suggests that we overweighed data from articulate and well-informed participants, and under represented data from less articulate ones (so-called elite bias [93]). We reduced elite bias through the inclusion of participants from many different domains and levels of experience. The reality of this research is that some *findings* are stronger than others and this is not dependent on the quality of the informants but rather our ability to find multiple instances that support our findings across multiple participants. Participants receive a roughly equal proportion of data in this thesis, e.g. the snapshot stories come from nearly all participants. When we gathered participants for our study, we consciously avoided bias towards local elite.

To extend the discussion of stronger or weaker data versus articulate or less articulate participants, consider the following case. We explored the hypothesis that participants who were stronger presenters produced more snapshots. We constructed evidence based on a formula that counted the number and types of drawn objects, and thus divided the participants into two groups – "strong presenter" and "weak presenter." However, when we made slight changes to the formula we used to pick groups, the hypothesis did not hold up. Thus, we did not present this finding.

A second case strengthens our argument. There were several times when we weighed the evidence with the purpose of finding negative evidence. Self-criticality is important, so when we devised our coding scheme and were looking for exemplars, we also looked for weak cases of codes so we could identify why they were weak and improve our coding scheme and deepen our understanding of code instances. The notion of weak snapshots also falls within negative evidence; we did find this category of snapshot, which did not fall within our coding scheme and did not make sense. In the end, we explored these cases in depth to understand under what conditions they exist and what the weak snapshots lacked; this helped us to improve our overall theory.

These two cases illustrate the general reality that we were more concerned with strong or weak findings and what to do with them than strong or weak participants.

T3.5: The principal investigator assumed the role of 'new-hire' during data gathering.

R3.5: We acted as new-hires during data gathering for two reasons. The first reason, because prior field studies and work experience indicated that communication with new-hires using a whiteboard was a common and important activity. The second reason was so the researchers could reasonably understand the session content during analysis.

What we did was not 'arms length research' and therefore, an important question exists: to what extent did the researcher questioning or providing feedback trigger snapshot behaviour? An examination of the protocol reveals minimal interaction (0.6% of the overall protocol) from the interviewer (the principal investigator) in which the interviewer asked the participant to reiterate what they said or asked for clarification, clarified why the participant exhibited certain non-uniform behaviour (e.g. using a different coloured pen), provided reminders regarding the time limitation of the study, and provided other non-verbal indicators such as smiling and nodding. The challenge of this question is this: by quietly listening, without interruption, or by certain unconscious non-verbal indicators, the researcher may have provoked snapshot behaviour. In future studies, we can improve upon our work by training an interviewer to perform the interview sessions so as to minimize this bias. We could then analyse the style and characteristics of the interviewer and assess the variability and impact on snapshot behaviour.

T3.6: Whiteboard sessions are not a fundamental aspect of software development.

R3.6: To address this threat we will provide evidence we gathered during post-session interviews with our participants. Our evidence suggests that software developers communicate using the whiteboard as a regular software development activity and that whiteboard sessions are indeed a fundamental aspect of software development.

Before we present our evidence, one may question if expert software developers give the majority of informal whiteboard sessions. When communicating with whiteboards, software developers have no predilection towards developers of particular levels of

experience. Every participant in our study regularly engages in whiteboard sessions. Whiteboard communication is exemplary and important.

Ten of our twelve participants endorsed the regular use of the whiteboard, with such statements as "crucial to model the more useful parts of software – in particular, the relationship between objects," "very useful" and "very useful in a formal context." One participant said he used the whiteboard "all the time, every day." Some participants indicated the consistent use of whiteboards for every design discussion, technical meeting, design meeting, and discussion with testers. Another participant described the update issue with diagrams in obsolete documentation; he pointed out that software explanation with diagrams is extremely useful as a resolution to this problem. One participant indicated that he communicated primarily with customers using PowerPoint; a second participant, however, indicated that "whiteboard diagrams are moderately useful, but no substitute for getting into the system."

Frank Wales, in a recent Internet Posting [143], also spoke to the fundamental nature of whiteboards:

> "These days, I don't believe I would hire a software developer who couldn't stand at a whiteboard and expound on a topic of interest for five minutes. As far as I'm concerned, it's up there with talking to clients on the phone or sending coherent e-mail messages. Communication with peers, clients, users and managers is critical to being able to do a good job in computing..."

T3.7: The work relationships among participants may affect how they draw software representations on the whiteboard.

R3.7: In response to this threat, we reiterate that a key goal in field research of this time is to *disturb the site as little as possible.* Aside from cursory details from some participants as to whom they work with, we had limited details of the participants' working context, e.g. the company product and their specific role. These details were insufficient to study whether the context affected how they used the whiteboard. To address this threat, we could gather more detailed accounts of the participants work agenda and specific details of their day-to-day work surrounding the study (as in ethnography, e.g. [61]), but this would greatly disturb the site. The downside is that without said context, we are unable to

determine which contexts are most related to our research questions. We can offer that the context we studied produced reasonable results and was therefore, in the very least, related to our research questions.

T3.8: There is dependence between results and the selection of the whiteboard as the medium. Why did the researchers not perform a study using modelling tools? Why did the researchers restrict access to resources such as PowerPoint or Word during whiteboard sessions?

R3.8: We specifically chose the most flexible communication medium with the idea that a tool developed as a result of the research would not end up being biased by artificial inflexibility derived from the research medium. If we are biased then we are biased in the direction we intended. The reason is that a tool constrains the user with the features the tool provides. Time also limited our study of alternative tools. Though the whiteboard does not provide access to the code or to existing models (e.g. for rationale), we do not have evidence to suggest that developers use source code or models during informal whiteboard sessions – they limit discussion to the board space during these sessions. Another issue with whiteboards is that participants do not have enough space to keep all their diagrams. However, because developers could freely add or remove elements, space was never an issue.

T3.9: Tool development or patterns will be biased towards whiteboards.

R3.9: According to 11 of 12 participants[28] in our IBM study, tools with features that resemble a whiteboard may be useful. These features include: the ability to design as one would design on paper, i.e. free-hand drawing features like those found in a Tablet PC; built-in collaboration; and the whiteboard as a capturing tool that takes a picture, which need not be syntactically correct, and converts it to a model.

T3.10: Study questions introduce bias towards a particular perception of architecture.

R3.10: We devised the study questions with managers at Mitel in order to develop stimulating questions that contained elements of architecture that are in widespread use and would promote the participants manipulation of their mental model. We placed a lot of

[28] The one participant who did not feel such features would be useful was knowledgeable of class diagrams and use case diagrams, but was a beginner or unfamiliar with other software diagrams.

emphasis on the first question and on architecture in general, and drilled down to a particular perception of architecture after the participant a) had already gone down that path or b) was having a hard time determining what we meant by architecture. We did not exclude any participant's interpretation of architecture that differed from our study questions.

T3.11: The researcher produced an effect on the study. The risk is that this behaviour can lead to biased observations.

R3.11: During our study, we tried to avoid the effects of the researcher on the case and the effects of the case on the researcher. The presence of a researcher inevitably produces some effect, and our goal is to clarify and reduce this effect.

The effects of the researcher on the study can manifest in many forms. Participants may switch their outward presentation to the researcher and craft their answers to protect their self-interests. To address this threat, we set the stage for the interview (see Section 4.2.2) with an introduction of the researcher, our motivation for the study and how data will be collected and handled. Also, our participants could not anticipate the study of snapshots as our core focus, so it is highly unlikely that they shaped their answers to affect our data. The high level of autonomy we gave participants through little interjection from the researcher further reduced researcher effects.

We minimized the effects of researchers on the study by:

- Staying on-site for as long as possible (three days a week for a year and a half) thus blending in to the environment;
- Using unobtrusive measures during study sessions (single camera, sessions of reasonable length, minimal impact on the site); and,
- Informing participants of our intent. In recruitment emails, face-to-face conversations and the introductory explanations to our study, we stated our motivation, the object of our study, our data gathering techniques and our data storage techniques.

We found some evidence to support a contention that the researcher had an effect on the process. For example, one participant indicated that he would have preferred more time to prepare and would have produced or used PowerPoint slides. This participant then gave

many short answers to the questions and produced very little diagrammatic material. There were occasional instances when a participant would get stuck, and rely on the researcher to guide the process. We have stated that our intent was to leave control in the hands of the participants and to use the open-ended questions to promote discussion, but in these instances, we guided the participants to encourage further discussion. During analysis, we were watchful for moments where the researcher promoted snapshots that would not otherwise occur and we did not count them.

As future work, we could address this threat by asking participants to review their own video and to ask if they feel their answers were contrived. To produce data with less researcher effect, we could extend our study to record ad-hoc informal whiteboard sessions, in particular, sessions that involve new-hires. To perform this step properly was beyond the scope of this thesis.

## 7.4 Reliability

Reliability demands a consistent study process across researchers and therefore, in our opinion, calls for systematic coding and an inter-rater reliability mechanism. As Judith Good [69] explains, "schemes which are not fully worked out and/or which are not accompanied by explicit instructions enabling them to be used by persons other than the original researcher are not of much use: it is impossible to compare results reliably."

T4.1: Results are biased and contrived.

R4.1: Multiple coders worked from a common coder's manual. We trained coders with example transcripts, coding examples and mentoring. Different researchers independently arrived at the same results using the same method. Coding checks were performed and showed an adequate level of agreement. Furthermore, reaching agreement improved coding in future iterations. Quality checks were performed regularly. We used peer review with a team of psychologists to strengthen the analysis process. In Section 4.3, we describe the inter-rater reliability process in greater detail. All in all, this helped us to avoid interpreting events as more patterned than they actually were. We did not shy away from weak data.

T4.2: Too few / too many categories

R4.2: We used a hierarchical structure with interdependent levels of granularity. Some categories were unused because they became obsolete (e.g. see codes change from Chapter 3 to Chapter 4). The codes allowed comprehensive coverage of the complete data set. There is no question that future work will yield new categories and discoveries as we work with new samples and our knowledge matures. For the purpose of this study, the number of categories was adequate.

T4.3: Researchers required hindsight to accurately code the dialogue, that is, the review of entire utterances in order to understand the meaning of the utterance was required to determine appropriate code and relationships to other codes.

R4.3: The risk with this threat is that too much iteration to properly code a chunk of data means that new coders may not be able to grasp the procedure at a glance. But our response is that experience comes from coding, and the decision about the appropriateness of codes was so regular as to become an essential part of the research process. Since we did not record all of the debates over coding decisions much of the rationale remains with the coders. Two factors come to mind: first, coding rationale is transferred during training and further code discussions; second, we updated the coder's manual with improved descriptions of code criteria. Multiple passes to properly code a chunk of data along with detailed discussions about the correctness or appropriateness of codes are intrinsic to qualitative research.

## 7.5 Objectivity

Objectivity in this context refers to the relative neutrality of the research and reasonable freedom from unacknowledged biases and explicitness about inevitable biases that exist [93, pp. 278].

T5.1: Data is not available for reanalysis by others.

R5.1: Our protocol data is confidential and so we cannot release this data. We provided a number of data displays in this research from which we derived our results. We also provided a number of vignettes to support the existence of our codes.

T5.2: The data is not precisely coded.

R5.2: Exactly when a participant created a new snapshot (according to our definition) was not always completely clear. Additionally our approach to identifying snapshots changed as we gained experience at coding the data. However, we reviewed every snapshot with multiple coders and focused our analysis intensely on snapshots. Moreover, drawing and speaking codes were precise to the second with correct behaviour. We therefore feel that this set of data is as precise as is possible within the time constraint of this dissertation.

T5.3: Inferences are drawn from non-representative events

R5.3: To combat the incremental slide into selective sampling and abusive generalizing, we critically reviewed and challenged our assumptions with a team of psychologists; in this way, we guarded against self-delusion.

We assumed that we were selectively sampling and drawing inferences from a weak or non-representative sample of people and challenged ourselves to prove otherwise. This was most evident when we began our analysis with just the first two participants: as exciting as it was to gain a deep appreciation of the two participants, and find rich (and contrasting) instances of snapshots, we knew we had to extend the number of participants until we found negative cases and new codes. And indeed, our sample was not wide enough until we had sampled roughly ten participants, at which point the number of regularities began to solidify. The final two participants confirmed our findings but did not yield significant variances to our findings. This led us to believe our sample was reasonably representative.

## 7.6 External Validity

Researchers building theories for practitioners are faced with the challenge of producing accessible, useful theories for practitioners to apply, and supplying data to reflect the process by which the theories were derived. The derivation process may not be useful to practitioners, but it is to other researchers who wish to validate or reflect on the theories themselves.

T6.1: How can we prove conclusively that the patterns we describe are an improvement on existing knowledge and therefore *are* a valid scientific advance?

R6.1: This threat suggests that there is no evaluation criterion for patterns. We submitted the patterns to the PLoP conference, where the patterns underwent a writer's

workshop and shepherding process. As a result, the patterns were further developed, expanded and rewritten more closely in the spirit Alexander intended; that is, we improved the problems, forces, applicability and overall quality of the patterns. We had numerous discussions with reputable pattern authors [5] who suggested that our patterns were well written, creative, useful and of a higher level of quality than typical patterns submitted to PLoP in the last ten years.

The issue is that this feedback does not provide evidence that the theories that underlie the patterns are an improvement on existing knowledge: there is no evaluation criterion for our empirical study. We believe that this statement highlights a key point for the whole thesis: subsequent generations of software engineers determine the value of a theory (sometimes many years or decades after the researchers form the theory). We can speak to how we applied methodological rigour, how we clarified our bias, how we assessed our threats to the validity of our methods and results. However, we cannot unequivocally determine that the theory we put forward is a contribution or scientific advance because future generations will determine this. That said, in the next threat resolution, we address the practical applications of our theory.

T6.2: Patterns are not at an appropriate level of abstraction. Concrete use of patterns is unclear. Effect of patterns on tool building and tool builders is not explicit.

R6.2: The risk is that if patterns are vague enough, people cannot disagree with them. As we built our theory, the concrete use of patterns was a significant challenge. We found many possible tool ideas. One well-developed idea to come out of this research is "The Diagram Player," an RSA feature currently in prototype development at IBM [48]. The Diagram Player captures the creation of a software model and allows for subsequent playback, capture of snapshots and annotation.

We should also note that Qanal allows for the capture, playback, and manipulation of snapshots from video-taped software explanation. Qanal will be useful as an educational tool in many contexts, including industrial training and university classrooms.

T6.3: Study uses analysis instrumentation that is not applicable in other study contexts.

R6.3: As software developers, we are privileged to be able to build tools to support the analysis process. As future work, roughly one person-month of the coding time is required to generalize the Excel macros for broader use in any qualitative study. Qanal is already usable for other studies because we use a generic XML schema.

## 7.7 Threat Summary

In this chapter, we described a number of threats and provided resolutions. We described threats to objectivity, reliability, internal validity, external validity and an overarching threat we call the explanation-cognition rift. Though our resolutions were sometimes built-in by design, some threats suggest opportunities for future work. Despite the threats listed in this chapter, the following points nevertheless suggest that our work has validity:

- We used multiple coders and investigated the agreement between them;
- We applied a well-founded analysis technique (grounded theory);
- The generation of our data displays provided additional validation checks;
- Many practitioners confirm that snapshots in theory and in practice are useful for discussion and constitute an improved design technique.

We continue a discussion of the contribution of this work in the next chapter.

# Chapter 8 Discussion[29]

## 8.1 Executive Summary

In the earlier chapters of this thesis, we provided a detailed account of an empirical study of software developers explaining software during informal whiteboard sessions. The study was designed to ground a theory of software explanation. We provided this theory in pattern form to facilitate the development of software tool features. Finally, we assessed the threats to the validity of our data, method, and results.

In this chapter, we summarize our contributions and discuss some implications of cognitive pattern research. Our primary contribution is a Snapshot Theory that describes how software engineers use snapshots to build the discourse structure through which they explain software. In the following sections, we review the theoretical and practical sub-contributions arising from our work. At various points in the discussion we provide a forward-looking perspective.

## 8.2 Theoretical Contributions

The theoretical contributions we make are Snapshot Theory and the Temporal Details Framework. Software engineering stands to benefit considerably from theories that help developers produce better tools and processes. Suggestions for such tool and process improvements are given below, after a recap of the theories.

### 8.2.1 Snapshot Theory

The primary contribution of our work is a theory of snapshots. This theory suggests that when explaining software, professional software developers use a series of "snapshots" with particular characteristics and relationships. Our data suggests the existence of a specific snapshot that conveys local details and a complete snapshot – that is, a combination of snapshots – that embodies global details. The condition for any kind of snapshot is a sufficient diagrammatic representation linked with sufficient structural or functional meaning.

[29] We deliberately use the term discussion. An exploratory study often leaves room for much discussion and little conclusion. The research described is an important first step, but conclusive results may take many years.

Our theory describes how a typical participant flows from an *infrastructure* snapshot, which is always found to be present, to an optional *advanced infrastructure* snapshot or *functional* snapshot and then to an *example* snapshot. Sometimes, however, the participant is unable to construct an appropriate snapshot in the absence of sufficient explanatory structural or functional material; the result is a *weak* snapshot.

We examined a process that developers regularly use and we analysed and conveyed the structure of software explanation with the goal of extending software modeling tools to support this behaviour. Our behavioural analysis of software explanation has led to results which explain how software developers systematize and express their knowledge structures when explaining software through speaking and drawing. This theory is therefore beneficial to the program-comprehension community: As prior contributions of program comprehension processes have shown, meaningful practical applications of the theory will follow. A comprehensive description of the temporal process of software explanation sheds light on how software development tools can better support the creation of software models. In Section 8.3.4, we describe the practical benefits of such a description.

The approach demonstrated in this thesis leads to the emergence of theory from an essentially open-ended research question. Many additional ramifications of this theory may be generated by the replication of the study in different contexts, and by further analysis of the data. For example, a great many different events in the data could be coded, potentially leading to many more cognitive patterns.

The second-order theoretical contributions to arise from Snapshot Theory are the concept of the snapshot itself, our particular categorization of snapshot categories, a specific set of propositional claims about snapshots, and the notion of the snapshot-event network. Each of these is summarized in the following paragraphs:

The concept of the snapshot originated in our interview sessions with professional software developers and evolved during our qualitative analysis. Our participants paused after key concepts and their explanation at that point contained dynamic details that were not apparent in their final modelled representation. These pauses marked coherent steps in the process of evolving a model towards its final form. In our analysis, we found the snapshot to be the culmination of an interval of time that contained enough information

about a model to reveal meaning. As we took the concept further, we recognized that a snapshot marks moments of conceptual wholeness at a given level of generality. That is, the snapshot marks a moment of insight where the meaning of a software model is evident. We realised that the snapshot moment could not occur until a participant revealed a sufficient amount of information. We also realised that the snapshot, irrespective of the meaning of the model it conveys, could reveal nascent information on its own or in conjunction with other snapshots.

To share the concept of snapshots with other researchers, we produced snapshot definitions, typical snapshot indicators and snapshot stories to describe the conditions, context and behavioural strategies we found in our research. Coders can use the definitions and typical indicators to examine discourse and identify the category of snapshot under observation as well as to know when a snapshot has occurred. The snapshot stories reveal the origin and composition of the discourse structure that leads up to the snapshot. The concept of a snapshot does not need to be limited to a qualitative analysis of software explanation; we hypothesize that tagging snapshots for education or presentation purposes can yield significant improvements in communicating software concepts. As future work, we will perform additional studies to deepen our understanding of these categories and the relationships between the categories.

The propositional claims arose during a qualitative analysis of software explanation in two development and maintenance domains with 24 participants. We contend that the following claims merit further investigation in further studies:

- Software explanation begins with and relies upon structure; basic structure is critical for software developers' understanding of a software model.
- In many cases, the basic software architecture provided with the infrastructure snapshot will not suffice and software developers require deeper structure in order to understand a software model.
- Functionality is more complex than structure.
- The participant produces the weak snapshot when the topic for explanation is outside the bounds of the participant's expertise and when descriptive meaning is replaced with extensive general discussion. The weak snapshot does not adequately

address structural meaning and does not provide an overarching description of the software model.

- Software professionals that produced more snapshots instigate more lateral discussion and produce a greater proportion of functional insight.

The branches in a snapshot-event network (e.g. Figure 4.18) reveal the participant's structural organization of key knowledge units with assigned priority. This network reveals how snapshots function within an entire explanation and how snapshots contribute to other snapshots. We found that the networks provided us with insight into overall explanation for each participant. Moreover, we used the networks as one means of checking data integrity. In future work, we will attempt to determine whether a snapshot-event network corresponds to the structure of knowledge the participant intended. We will also further explore the rationale for the transition between snapshots. We also believe it is useful to compare the networks built by multiple researchers; this form of inter-rater reliability checking would help ensure that networks are unbiased and representative of the data. Snapshot-event networks were the link from our Snapshot Theory to our Temporal Details Framework, the contributions of which we describe in the next section.

### 8.2.2 Temporal Details Framework

*Temporal details* are a network of snapshots, composed of one or many sequential snapshots and one or many parallel snapshots. That is, snapshots are the building blocks of temporal details. Temporal details also comprise a high-level pattern whose forces are resolved by several other patterns. The audience for the *temporal details framework* are software developers involved in the maintenance of large-scale legacy software, or tool developers who build tools to aid such maintenance. By using a tool based on these patterns, a software practitioner may be able to locate undocumented design decisions and, as a result, understand the system in its current form by understanding how the system evolved over time.

Tools supporting the collection, composition and manipulation of snapshots will support temporal details and thereby help developers, by understanding the system's history, understand why the system is the way it is today and the constraints upon the system. Because historical information may be exceedingly rich and complex, if we are to

improve understanding, we must incorporate the capacity to capture snapshots, compose snapshots into networks, manipulate these networks of snapshots, and visualize the networks. The practical implications we will describe in Section 8.3.4 include tool features for explaining, exploring and documenting systems that mesh more closely with the way users think and act. The focus of this thesis has been to understand snapshots, and future work should deepen the understanding of how collections or networks of snapshots are constructed and are useful in practice to software developers.

## 8.3 Practical Contributions

In pursuit of the theoretical contributions, we found it necessary to build a number of practical qualitative research contributions including our qualitative research methodology, our data, and our qualitative analysis tool, Qanal. Other software researchers engaged in qualitative software engineering research may readily apply these contributions. Also, our cognitive patterns constitute a practical contribution that tool developers may refer to when building tools.

### 8.3.1 Qualitative Data Analysis Research Methodology

We presented a rigorous qualitative data-analysis process (c.f. Figure 4.1) that led us to transparent[30] results that are rich, descriptive and closely linked to the data. The process involved the repetition of three core operations (description, analysis and interpretation) as we stepped through five main processes (data preparation, initial analysis, topical analysis, category analysis, and patterns & theories). The analytic operation was based on the grounded theory approach as extended by Miles and Huberman [93]. The contribution of this methodology is significant and offers exemplary methodological processes through practical challenges.

We suggested how to construct and administer interview sessions when the objective is the generation of open-ended data. We condensed many data sources from interviews by tagging categories of meaning using a set of software explanation codes. We further reduced our data by constructing a series of tables and charts that allowed us to better

[30] The challenge in presenting 'transparent' results is that compiling the complete data set for publication is not possible on account of space constraints, participant confidentiality and corporate confidentiality. We therefore use the term 'transparent' to refer to results we found by applying transparent process steps, described in detail in Chapter 4.

understand data trends or patterns. We built a set of claims based on these patterns. We then traced the patterns back to the source data to ensure the existence of an appropriate trail of evidence and to discover the source context in which our findings exist. This deepened our interpretation of our findings and helped to produce a parsimonious theory.

Much of Chapter 4 described the practical application of our process methodology. The detailed steps are an improvement over the prior comprehension process studies described in Section 2.4, which do not indicate specifically how they found the developers' comprehension processes. Software researchers can draw upon our experience to overcome challenges such as:

- gathering data (the practice of not disturbing site, theoretical sampling, forming open-ended interview questions to answer research questions, dealing with video data)
- developing and handling codes (including entering data, data syntax, avoiding preconceptions, code evolution, code obsolescence)
- emerging study design (conducting a pilot study, evolving protocol, identifying central phenomenon, refining coding scheme, continual reflection, using the constant comparative approach)
- analysing coded data (reduction, sorting and rearranging coded sequences, providing rationale in annotation, generalizing a small set of propositions, tactics such as noting patterns, plausibility, counting, etc.)
- working with discrepant evidence
- assessing inter-rater reliability and working with multiple coders.
- understanding saturation
- building matrix and chart displays
- identifying patterns and developing theories
- performing category analysis

Other improvements over prior work include our interview structure, devised to regularly prompt inquiry moments and permitting us to receive the richest possible response. Furthermore, our work investigated a task that is consistent with our participants' daily agendas, regular communication activities and natural environment. We also noted

that 25% of our total sessions were spent sketching, a fact that indicated the significance of the sketching activity in software explanation. Among published software research studies investigating comprehension processes, this is the first to include the sketching aspect of software explanation.

We recognize the need to experiment with further methodological improvements: One possibility would be to examine the emergence of snapshots in truly informal whiteboard sessions without researcher involvement (aside from, for example, a hidden camera). This triangulation by method would help us examine if snapshots in an informal context are responsible for better explanation or understanding.

Another possibility would be to examine how snapshots promote understanding in new-hires and therefore the effect complete snapshots have on an audience. In such a study, we could mine snapshots using a similar method and examine them from the perspective of a real new-hire. This is also triangulation by method.

In a third study, we could invite participants to validate their own or someone else's snapshots. This member-checking approach would help us to study the participant's intent to produce complete snapshots. We did not apply this step in our present research because:

- We did not want to introduce bias.
- We did not wish to disturb the site more than necessary.
- We felt participants may not be consciously aware of how or why they form snapshots.
- We were under time constraints.

In a fourth possible study, we could perform triangulation by data source in a cross-sample comparison of our Mitel, IBM and perhaps a third independent data sample. It might be worthwhile, for example, to contrast our findings with snapshots derived during business process modelling or other diagrammatic techniques from other domains than software.

In a fifth study, we can illustrate *only* the snapshot material to participants and study software understanding and characteristics of snapshots from the participants' perspective.

In a sixth study, we could invite participants to build models using a tool (such as RSA) and then we can gather confirming evidence along the lines of our research. A researcher in our research lab is building a tool that will support this research step [48].

### 8.3.2 Qualitative Analysis Tools

We built instrumentation to aid our qualitative data analysis efforts. More specifically, we wanted the frequent task of the capture and replay of video analysis to be repeatable and easy. We built a Java-based tool, Qanal, for analysing codified video data. We imported XML files of coded data into Qanal. We call each file an exploration. Qanal supports VCR-like playback of the videos that correspond to the explorations. The collection and manipulation of explorations allows a researcher to review similar groups of data to confirm hypotheses, to review the analysis history, or to share analytical findings with other researchers.

Other software researchers engaged in qualitative research with coded video data can use Qanal to simplify and extend their analyses. Qanal is platform-independent, can be used with a long list of video types, and is not tied to a particular coding scheme.

Qanal is applicable beyond qualitative analysis. The navigation and sharing of codified video data that is tied to corresponding image files can be used in pedagogical contexts.

Researchers may also be interested in using our Excel instrumentation. We built powerful tools for the generation of chart and matrix displays. The benefits of our Excel instrumentation include the integrity checker, data modeller, and display builder; building displays by hand is labour-intensive and error-prone. We also designed a macro to export our coded data to Qanal. The generation of chart and matrix displays is tightly coupled with our data model, the participants and the coding scheme. In future work undertakings, we will generalize our approach to work with any coding scheme and any number of participants. We will also package the source code for wide-spread use.

### 8.3.3 Case Data

Researchers may be interested in building on such case data as our coding scheme, displays, propositional claims and snapshot story boards[31]. The coding scheme is still relevant and extensible for another study with a different central phenomenon. One benefit of applying our scheme is easy adoption of our analytic tools. We intend to make our analytic tool generic to any coding scheme.

Our coding scheme underwent significant evolution to reach its current state. However, by the end of our data analysis, it was evident that our coding scheme was sufficient to codify the software explanation aspects of our entire data set. The evolutionary changes to our coding scheme were quite costly in terms of time. Further studies that apply our coding scheme will inevitably improve the code descriptions.

Other researchers will find that the range of displays we built in conjunction with a rigorous analytic process may yield new insight into difficult software engineering problems. The contribution includes:

- The *time-ordered matrix (Figure 4.12)*. This matrix may seem intuitive, but it came about through much trial and conveniently handles data. More specifically, reading codes, manipulating codes and building a data model were made easier.
- *Drawing-speaking independent matrices (Table 4.1)*, for which we described how to interpret code counts in the context of the participant, the study, other code counts, durations or proportions.
- *Drawing-speaking relationship matrices (Table 4.3),* for which we describe how meaning enriches diagrammatic material and how we linked explanation activities with knowledge concepts.
- *Drawing-speaking-snapshot relationship charts (Figure 4.13),* which we used to note relationships among codes and to find patterns, specifically related to snapshots.

[31] We cannot provide our entire set of case data on account of participant confidentiality, corporate confidentiality and space limitations.

In future projects, we aim to produce a word-frequency analysis that aligns a model of the user's knowledge with our snapshot event network. This will provide the role of the snapshot in the developers' cognitive model.

### 8.3.4 Cognitive Patterns

A cognitive pattern is a structured textual description of a solution to a recurring cognitive problem in a specific context. Cognitive patterns bridge the gap between theory and application in that they are discovered in field research and can be applied to help derive features of tools that aid in understanding software, whether for purposes of design, or some other type of problem solving. Such features will support such cognitive activities as reasoning about and thinking about software artefacts.

The application of the *temporal details framework* and its related patterns will result in tools for explaining, exploring and documenting systems that take advantage of the knowledge embedded in a model's history and that mesh more closely with the way users think and act. For example, a complete applied design of temporal details would support the ability to manipulate history. The applied design of snapshots would enable users to build and present a model in appropriately sized increments, and allow the user to reference and come back to some of those model versions if needed. The applied design of long views would enable the explanation and exploration of the history of a model through sequences, rather than just a simple disorganized presentation of snapshots. The applied design of multiple approaches would support the users' designation, exploration and visualization of multiple paths for understanding, comparison and learning. The applied design of manipulate history would support the manipulation of networks of snapshots by designating points, sequences and branches in the history of a model's evolution and removing history that does not enhance comprehension. The applied design of the meaning pattern would support annotations of evolution rationale (e.g. why new details are added or replaced to create a new snapshot.)

Tools currently tend to be designed with the final diagram in mind; if the tools were designed with the evolving state of a representation (including deletions) in mind, it may be possible to improve their user interfaces. The bottom line is that we are using these patterns as generalizations that should be useful to tool designers building actual tool prototypes.

We expect feature improvements along the lines of design rationale, annotation and traceability. We also expect future work to examine the educational component of software explanation through temporal details. The known uses of the cognitive patterns we have described are limited instantiations of our patterns and can be improved in future work. By improving our known uses, we can improve tool designers' ability to design tools.

The application of the temporal details patterns we will examine in the near future is snapshots in support of software walkthroughs. In this case, the snapshots can be the actual system or the representation of the system. Improvements to the process of walking through a dynamic software system will yield tremendous insight into the benefit of temporal details from application and documentation perspectives.

It is worth noting the relationship between our research and case-based reasoning (CBR) [64, 117]. CBR is a human behaviour in which a human matches a new problem with problems from a bank of old solved problem in order to derive a new solution from solutions to the old problems. One might argue that all we do as humans is case-based reasoning; we just store cases and then match these cases in new situations. The AI community has taken an interest in modeling this behaviour and has developed technology that supports the retrieval, analogical matching, adaptation and learning of cases. There are several analogies between our work and CBR:

- In grounded theory, we are looking at and encoding a small number of sessions (each session is a case).
- The snapshots we find in the video are cases we might find in a particular design
- The archetypical snapshots are generalized cases we used to match new instances of snapshots.
- In the library of temporal details patterns, each pattern is a generalized case for reasoning. In fact, all patterns are just cases!
- Our participants used case-based reasoning to match the problems they faced at the whiteboard with older problems they were more familiar with.

Our cognitive patterns serve as the first steps towards what might be called, a "handbook of software comprehension." The patterns offer solutions, or steps to generate solutions, to practical problems in cognition and software comprehension. Our patterns

have endured community scrutiny through both vigorous shepherding and writer's workshops. The notion of empirical studies that reinforces pattern development corresponds to the need for firm grounding in well-received theories from cognitive science and HCI described by Walenstein [142] in addition to the call for empirical grounding expressed by Cross *et al.* [34].

Tool development based on the patterns has already started [48]. We hypothesize that tool developers may use cognitive patterns to understand their users and cognitive patterns may actually improve the tool developers' own mental models; we propose studying this as future work.

In general, we wish to create the infrastructure for a patterns community that could emerge in the program comprehension field, one that recognizes the specific concerns of this field. This infrastructure would include the notion of cognitive patterns, pattern languages, shepherding [60] and writing workshops [111] — all adapted from the pattern community at-large. In fact, cognitive patterns, as we envision them, may be applicable in other domains as well, such as the creation of new software, or the collaboration of groups of people in teams; however, our focus remains the way in which people understand software.

## 8.4 Concluding Remarks

This thesis represents a complete snapshot of our research. Indeed, this work underwent much evolution and many historical changes: many snapshots preceded this 'final snapshot.' To produce this complete snapshot, we reduced and manipulated the snapshot network of our research. The network of snapshots that remains annotates only the evolutionary steps that provide comprehension of our theory, our methodological process and our findings, in the form of cognitive patterns. We opened up a research area for numerous future studies. In these early stages of snapshot research, we established a terminology and a structural foundation to support the future work. We anticipate that the future of snapshot research and cognitive pattern research will leverage the patterns expressed in this study to gain a deeper realisation of their benefits in tools yet to be developed.

# Appendix A: Role of the Researcher and Coder Profiles

## Role of the Researcher

Qualitative research is interpretative research which brings a range of strategic, ethical and personal issues into the research process [86]. We explicitly note the principal researcher's biases, values and personal interests so the reader attains better perspective of the research process and results. The researcher's contribution to the research setting can be useful and positive rather than detrimental [86]. I will use first-person rhetoric to relate my personal experiences, my aim, to relate my unique contribution to this research.

I worked two years at Alcatel, in a software support lab for an ATM switch – an analogous situation to our study participants'[32]. 'Stepping into the shoes' of the developers at Mitel was straightforward; the context of the participants was second-nature given my experience. Also, my personal values and beliefs with respect to software maintenance were comparable to those of our participants – the study setting had a culture not unfamiliar to me, and collaboration with participants was uncomplicated. One of our original concerns was one of conflict of interests between Mitel and Newbridge – non-disclosure and contractual agreements with both companies established the groundwork to proceed with research.

My research began roughly three years after initial relationships were established between our research group, the KBRE, and Mitel. Site and participant selection was relatively uncomplicated and involved sequestering management support and planning for minimal disruptions to employee workflow. Our initial plan was to provide Mitel with a set of synthesized diagrams they could use for new-hires. The challenging nature of qualitative research for the novice researcher strained the research process, and the decrease in R&D spending as a result of the telecommunications melt-down dissolved our partnership before we could provide results. The disbanded relations had an adverse effect on collecting further participant meaning. However, our videotaping approach made this effect limited, perhaps even negligible.

[32] Incidentally, Terry Matthews started both Mitel and Newbridge – they are two blocks apart in Kanata, Ontario. At a glance, the similarities between the companies were startling.

I offer the following simple lessons. Do not underrate the trials of grounded theory research. Software engineering presents cultural gaps. Do not underestimate the value of an industrial partner. From a pragmatic perspective, this is more than funding; the value comes from collaboration with talented and like-minded people. These relationships form the possibility for creative and innovative research, connect academic research with the "real-world", and allow one to face the trials of research with vigour. The lessons are important to the researcher who intends to embark on a similar path.

**Coder Profiles**

As described in Section 4.3, we used multiple coders to analyse our data. In addition to the principal investigator, five other coders reviewed at least part of the data. All of the coders enhanced our understanding of the coding scheme and our protocol data as a result of numerous discussions surrounding the methodology and data.

Cognitive Psychology Coders: Two of the coders were doctoral candidates in cognitive psychology and had experience with qualitative research. Also, one of these coders had experience specifically in the application of grounded theory in an industrial research environment. Neither of these two coders had any prior knowledge of professional software development, though they commented that they benefited immensely from the opportunity to learn more about this domain which has been a subject of interest in cognitive psychology for some time. They furthermore commented that they understood the software concepts with further review of the data, at least from a high-level perspective. The benefit of their insight came in discussions surrounding both our methodology and also the notion of the snapshot as a moment of insight in the explanation of software. These two coders performed multiple passes over the complete data set. A third coder from the cognitive psychology program (unsuccessfully) coded a single participant's data but the level of understanding regarding how to code data and inability to understand software concepts from a high-level (and therefore an inability to understand and properly code the explanation) made this candidate unsuitable for coding data.

Software Coders: Of the two remaining coders, one hails from an academic background having completed a Ph.D. in Computer Science. In addition, this person has practical experience with empirical studies in professional development environments, and

modest experience coding data as in grounded theory, as well as programmatically coding complex real-time software systems. This coder coded only a single participant. The other software coder has a college diploma in computer programming and is therefore fluent in the software concepts described by our participants. This coder very quickly adopted the coding scheme and in general was able to work through the data at a quicker pace than the cognitive psychology coders. This coder performed a single pass over the complete data set.

# Appendix B: Snapshot Storyboards (conditions, context, strategies)

This appendix provides example snapshot storyboards. We analysed the storyboards (c.f. Section 5.2) for each snapshot category to gain insight into *commonalities and differences* within and among snapshot categories, the *conditions* that contribute to the existence and occurrence of snapshots, and participant *strategies*.

**Infrastructure Snapshot (1[33])**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 0:42 |
| # Model Elements: | 3 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 3 sec |

**Conditions:**

| | |
|---|---|
| ADD: | 3 boxes |
| MEAN: | 3 |
| TALK: | 0 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 0 |
| ADD Duration: | 0:29 |
| MEAN Duration: | 0:13 |
| TALK Duration: | 0 |
| Links within zone: | 1-1; 1-2 |
| Loners: | 1 |
| Loner duration: | 0:05 |

**Context:**

TALK-D: 0

**Strategies:**

This infrastructure snapshot starts with a general description of a model through the use of layers ('*RSA is a layer that is built on top of another set of layers*'). The participant

[33] The number '1' refers to the participant identifier

gives a vertical order of components, which start from the bottom of the whiteboard and progresses upwards. The participant hints at a link between two components but offers no directional talk. The participant adds two boxes, explains their meaning, then adds a third box and explains its meaning. The participant offers no concluding rationale, instead moving towards the next advanced infrastructural snapshot.

**Infrastructure Snapshot (7)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 1:07 |
| # Model Elements: | 2 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 0:05 |

**Conditions:**

| | |
|---|---|
| ADD: | 2 |
| MEAN: | 3 |
| TALK: | 3 |
| TALK-C-SE: | 1 |
| TALK-C-OE: | 0 |
| TALK-D: | 2 |
| ADD Duration: | 0:05 |
| MEAN Duration: | 0:29 |
| TALK Duration: | 0:33 |
| Links within zone: | 1-1; 2-1 |
| Loners: | 0 |
| Loner duration: | 0 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 1 |
| TALK-D end: | 1 |
| TALK-C-SE beginning: | 1 |
| TALK-C-SE end: | 0 |
| TALK-C-OE: | 0 |

**Strategies:**

This infrastructure snapshot starts with a self-evaluative talk, which suggests less confidence and therefore less focused discussion in the participant's area of expertise. The participant then continues with directional talk to indicate what this area of expertise is, and thus how the presentation will proceed. The participant uses two horizontal labels to represent two components, with no links between them. The participant then writes one component's function on the whiteboard. The participant ends the snapshot with conclusive directional talk ('this is a basis of …').

**Advanced Infrastructure Snapshot (3)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 2:15 |
| # Model Elements: | 9 |
| # corrections: | 2 |
| Pause Duration following Snapshot: | 0:08 |

**Conditions:**

| | |
|---|---|
| ADD: | 8 |
| MEAN: | 6 |
| TALK: | 1 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 1 |
| TALK-D: | 0 |
| ADD Duration: | 1:20 |
| MEAN Duration: | 0:56 |
| TALK Duration: | 0:19 |
| Links within zone: | 1-1; 1-1; 1-2 |
| Loners: | 8 |
| Loner duration: | 0:40 |

**Context:**

| | |
|---|---|
| TALK-D, C-SE: | 0 |
| TALK-C-OE beginning: | 1 |
| TALK-C-OE end: | 0 |

**Strategies:**

This advanced infrastructure snapshot starts with an objective evaluative talk about how to present the drawings (*'In drawing this I've found that trying to make this huge thing with all these little boxes just turns out incomprehensible.'*). At the beginning, he hesitates to point to some imaginary *'tiny little boxes here'*. The participant then removes one side of the box and extends it. He links with the previous infrastructural model, explains the functionality of the components that create the first box, and then proceeds with building the second one. The participant does not establish links between the two boxes on the diagram. The snapshot concludes with general talk about the java package (*'But, there's not*

*very much in a java package, it is really just a directory structure, so this rule doesn't do very much. What's interesting is the class and interface transformations.*').

**Advanced Infrastructure Snapshot (10)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 2:34 |
| # Model Elements: | 3 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 0:02 |

**Conditions:**

| | |
|---|---|
| ADD: | 3 |
| MEAN: | 0 |
| TALK: | 1 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 1 |
| TALK-D: | 0 |
| ADD Duration: | 0:14 |
| MEAN Duration: | 0 |
| TALK Duration: | 2:20 |
| Links within zone: | 0 |
| Loners: | 3 |
| Loner duration: | 0:14 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 0 |
| TALK-D end: | 0 |
| TALK-C-SE beginning: | 0 |
| TALK-C-SE end: | 0 |
| TALK-C-OE beginning: | 1 |
| TALK-C-OE end: | 0 |

**Strategies:**

This snapshot starts with an overview of the invisible architecture ('*there's an invisible part to the architecture...*'), and continues with a long TALK (2 minutes and 20 seconds). The snapshot concludes with two lines of text added to the box. There is no conclusive talk at the end.

**Example Snapshot (2)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 3:19 |
| # Model Elements: | 11 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 0 |

**Conditions:**

| | |
|---|---|
| ADD: | 5 |
| MEAN: | 3 |
| TALK: | 3 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 0 |
| TALK-D: | 3 |
| ADD Duration: | 1-10 |
| MEAN Duration: | 1:20 |
| TALK Duration: | 0:31 |
| Links within zone: | 1-1; 1-1; 1-1; |
| Loners: | 2 |
| Loner duration: | 0:35 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 2 |
| TALK-D end: | 0 |
| TALK-C-SE, C-OE: | 0 |

**Strategies:**

The snapshot starts with a statement of assumption about the listener's knowledge ('*So let's assume you know what a class looks like when you draw it* ...'), continues with an extensive talk while drawing, and then states four directional questions and answers them. The participant states one practical problem ('*Now, we drew this (the list) on the screen, how does that work?*') after drawing an actor on the whiteboard, he steps back and asks another question ('*How do I create a request from this*?'). He makes a long explanation of element's functionality (00:01:15), with no conclusive talk at the end, but proceeds toward the next snapshot.

**Example Snapshot (6)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 02:25 |
| # Model Elements: | 7 |
| # corrections: | 1 |
| Pause Duration following Snapshot: | 0:04 |

**Conditions:**

| | |
|---|---|
| ADD: | 6 |
| MEAN: | 2 |
| TALK: | 1 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 0 |
| TALK-D: | 1 |
| ADD Duration: | 1:39 |
| MEAN Duration: | 0:35 |
| TALK Duration: | 0:11 |
| Links within zone: | 1-1; 1-1; |
| Loners: | 4 |
| Loner duration: | 0:57 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 1 |
| TALK-D end: | 0 |
| TALK-C-SE, C-OE: | 0 |

**Strategies:**

The snapshot starts with an objective evaluation about the existence of one editor that should be used, instead of propagating different versions. The participant states a practical question ('*The one issue could be how do you get to this, how do you make a query*'), and when asked to show some cases, the participant proposes going step-by-step in the presentation and at the same time shows uncertainty in the graphical display ('*I have to go step by step, for example, in this use case here, the user will first select...I don't know how to draw this, because*...). The participant then writes each step in text on the whiteboard (e.g., '*user selects WSDL creation tool and drops it onto the diagram,*'). The participant

then concludes with a summary ('*All these steps, you select, then the drop a few wizards, name, service name, all the details, then to finish this will draw, so this is a use case.*').

**Lateral Snapshot (5)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 1:08 |
| # Model Elements: | 4 |
| # corrections: | 1 |
| Pause Duration following Snapshot: | 0:05 |

**Conditions:**

| | |
|---|---|
| ADD: | 4 |
| MEAN: | 2 |
| TALK: | 0 |
| ADD Duration: | 0:22 |
| MEAN Duration: | 0:45 |
| TALK Duration: | 0 |
| Links within zone: | 1-1 |
| Loners: | 4 |
| Loner duration: | 0:47 |

**Context:**

TALK-D, C-SE, C-OE: 0

**Strategies:**

The participant develops a structural explanation of one of Eclipse's components, and the participant draws the structural elements of the diagram. The participant explains that some components are not used and limits his presentation to one aspect of the infrastructure (*'so I just put them there to show that there is a way to extend the Eclipse platform in a lot of different areas from reporting to testing to development environment.'*). The participant builds on top of the Eclipse components with boxes. The participant also links this diagram with previous drawings (C/C++ box which is the same as the one in a previous diagram – *'and if you remember the previous drawing we have a C/C++ box, we're just reusing the same box from the Eclipse foundation package'*). The participant also expands on other tools on the top of Eclipse by adding two more boxes. The participant removes parts that represent lateral components of the infrastructure. The key feature of this snapshot is the

focus on the supplementary information about the pre-existing infrastructure – the key reference to non-core information is critical.

**Lateral Snapshot (2)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 2:13 |
| # Model Elements: | 6 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 0:10 |

**Conditions:**

| | |
|---|---|
| ADD: | 3 |
| MEAN: | 3 |
| TALK: | 1 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 0 |
| TALK-D: | 1 |
| ADD Duration: | 0:35 |
| MEAN Duration: | 1:22 |
| TALK Duration: | 0:13 |
| Links within zone: | 1-1; 1-1 |
| Loners: | 3 |
| Loner duration: | 1:25 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 1 |
| TALK-D end: | 0 |
| TALK-C-SE, C-OE: | 0 |

**Strategies:**

The participant develops the lateral functional snapshot on the edit part through the provision of a forward-looking statement, a question, and an evaluation of the model's simplicity. There is directional talk at the beginning: '*Coming back to the model, whenever an element changes there are events that get fired, and in this case it will be semantic events, and in this other one it will be notational events*', and also '*how do we listen to it*?', and then '*it's very simple…*'. The participant draws a model (cylinder) on the whiteboard and adds a table with labels in it. The participant finishes with 58-second concluding talk

about the functionality of an edit part that underlines the essence of this lateral sequence of this presentation. The concluding remark is: '*So that takes care of the edit part.*'.

**Weak Snapshot (4)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 1:23 |
| # Model Elements: | 7 |
| # corrections: | 1 |
| Pause Duration following Snapshot: | 0:05 |

**Conditions:**

| | |
|---|---|
| ADD: | 6 |
| MEAN: | 1 |
| TALK: | 3 |
| TALK-C-SE: | 2 |
| TALK-C-OE: | 0 |
| TALK-D: | 1 |
| ADD Duration: | 0:52 |
| MEAN Duration: | 0:12 |
| TALK Duration: | 0:18 |
| Links within zone: | 1-1 |
| Loners: | 5 |
| Loner duration: | 0:49 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 1 |
| TALK-D end: | 0 |
| TALK-C-SE beginning: | 1 |
| TALK-C-SE end: | 1 |
| TALK-C-OE: | 0 |

**Strategies:**

The participant is quick to start, and provides a self-evaluation ('*I am trying to think because it has been some time before I was on this.*'). The participant continues by drawing structural components on the whiteboard ('*in terms of classes...*'). No functionality is explained, and no concluding talk is given. The participant finishes with another structural element '*I don't remember exactly what it looked like.*'). The participant added lines of text on the board, with one remove-practical, and with frequent use of expressions such as '*there were ...*', or '*we would have...*' The participant's content is incomplete at the end of

this sequence and the participant is reduced to a subjective evaluation ('*I am a bit fluffy here. I don't remember exactly what it looked like.*').

**Weak Snapshot (8)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 1:49 |
| # Model Elements: | 3 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 0:04 |

**Conditions:**

| | |
|---|---|
| ADD: | 3 |
| MEAN: | 1 |
| TALK: | 5 |
| TALK-C-SE: | 3 |
| TALK-C-OE: | 2 |
| TALK-D: | 0 |
| ADD Duration: | 0:33 |
| MEAN Duration: | 0:04 |
| TALK Duration: | 1-12 |
| Links within zone: | 1-1 |
| Loners: | 2 |
| Loner duration: | 0:25 |

**Context:**

| | |
|---|---|
| TALK-D: | 0 |
| TALK-C-SE beginning: | 1 |
| TALK-C-SE end: | 1 |
| TALK-C-OE beginning: | 1 |
| TALK-C-OE end: | 0 |

**Strategies:**

The participant starts with a self-evaluation – where the participant worked before and how much the participant knows about profiles (e.g., '*You'll have to take my answer with a bit of a grain of salt*,' and '*I don't know much about dynamic EMF*...'). He adds boxes such as 'e-class' and features such as 'foo', ... (he leaves one box without a label in it) and objects that are instances of a class. The participant conveys the present understanding of how profiles are used, and shows a lack of knowledge (self-evaluative talk) of how they function, '*How they persist, that info and how they keep it up to date I don't know*'.

**Functional Snapshot (6)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 2:04 |
| # Model Elements: | 1 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 0:00 |

**Conditions:**

| | |
|---|---|
| ADD: | 2 |
| MEAN: | 5 |
| TALK: | 0 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 0 |
| TALK-D: | 0 |
| ADD Duration: | 0:23 |
| MEAN Duration: | 1:47 |
| TALK Duration: | 0 |
| Links within zone: | 1-1 |
| Loners: | 4 |
| Loner duration: | 1:20 |

**Context:**

| | |
|---|---|
| TALK-D, C-SE, C-OE: | 0 |

**Strategies:**

The participant selects the Aurora platform plug-ins to explain in more detail. During the presentation the participant adds just a few lines, provides functional talk about plug-ins ('*so these are plug-ins, think in terms of plug-ins, they provide... ...some service.*'). He adds new services and explains their function on the fly. He provides no drawings, only functional explanations. The participant concludes with the notion of what every newcomer should know about the services provided by the Aurora platform ('*So, we use a whole bunch of service here we use, as a newcomer you need to know, not the whole service, but the service required to implement only this part. You implement using the platform, somebody can implement from the ground level*').

**Functional Snapshot (12)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 3:55 |
| # Model Elements: | 15 |
| # corrections: | 1 |
| Pause Duration following Snapshot: | 0:00 |

**Conditions:**

| | |
|---|---|
| ADD: | 9 |
| MEAN: | 9 |
| TALK: | 0 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 0 |
| TALK-D: | 0 |
| ADD Duration: | 1:28 |
| MEAN Duration: | 2:25 |
| TALK Duration: | 0 |
| Links within zone: | 1-1; 2M:1A; 2M:1A; 1-1 |
| Loners: | 8 |
| Loner duration: | 1:24 |

**Context:**

| | |
|---|---|
| TALK-D, C-SE, C-OE: | 0 |

**Strategies:**

The participant talks about the 'BPEL' editor and its functions. The participant adds boxes and explains their structure, adds text lines on the board with structural description, and splits the main display into left and right parts. The participant extensively uses the whiteboard to build a model with numerous boxes and labels on it. However, he gives no directional talk, and no conclusive talk in this part. The participant makes links with previously presented models. The participant frequently uses the expression '*we have*' this and that in the presentation. The only real functional part here is at the end of this section when the participant gives a functional explanation for the edit part (27 sec.). All previous explanations are structural (MEAN-S-C).

**Sequential Snapshot (9)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 9:01 |
| Type of SNAP preceded | L-func |
| # Model Elements: | 8 |
| # corrections: | 2 |
| Pause Duration following Snapshot: | 0:09 |

**Conditions:**

| | |
|---|---|
| ADD: | 3 |
| MEAN: | 5 |
| TALK: | 5 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 0 |
| TALK-D: | 5 |
| ADD Duration: | 0:55 |
| MEAN Duration: | 3:35 |
| TALK Duration: | 0:45 |
| Links within zone: | 3M:1A; 1-1; 1-1 |
| Loners: | 0 |
| Loner duration: | 0:00 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 4 |
| TALK-D end: | 0 |
| TALK-C-SE, C-OE: | 0 |

**Strategies:**

The participant constructed a lateral snapshot on the extensibility mechanism with the UML diagram core; the participant expands this discussion in this sequential snapshot. The participant states the question: '*What is a UML class shape*?' The participant asks how to visualize the UML class, and chooses the box form. The participant also asks a directional question about filling up the box while he draws a diagram (*'how do we know how to populate what's inside the box if we didn't have the semantic reference?*'). The participant

presents strong concluding talk for over a minute, describing the overarching function of the name and list compartments.

**Sequential Snapshot (2) (Please note, prior snapshot included in values below)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 4:29 |
| Type of SNAP preceded | Exam |
| # Model Elements: | 15 |
| # corrections: | 3 |
| Pause Duration following Snapshot: | 0:05 |

**Conditions:**

| | |
|---|---|
| ADD: | 6 |
| MEAN: | 4 |
| TALK: | 4 |
| TALK-C-SE: | 0 |
| TALK-C-OE: | 0 |
| TALK-D: | 4 |
| ADD Duration: | 1:58 |
| MEAN Duration: | 1:52 |
| TALK Duration: | 0:33 |
| Links within zone: | 1-1; 1-1; 1-1 |
| Loners: | 5 |
| Loner duration: | 1:02 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 3 |
| TALK-D end: | 0 |
| TALK-C-SE, C-OE: | 0 |

**Strategies:**

The participant provides a sequential snapshot built on a previous example snapshot. The participant returns to diagrammatic material and starts with a directional question ('*you want to create something, so on the palette tool ... what do we do?*'). The participant creates a large box with smaller boxes and a circle in it to represent the structure of an application interface. The participant produces a sequence diagram. The participant concludes with strong directional talk about where the explanation is going (36 seconds).

**Complete Snapshot (1)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 3:59 |
| Type of SNAPs preceded | Infra, advinfra, L-func |
| # Model Elements: | 13 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 0:02 |

**Conditions:**

| | |
|---|---|
| ADD: | 7 |
| MEAN: | 7 |
| TALK: | 2 |
| TALK-C-SE: | 1 |
| TALK-C-OE: | 0 |
| TALK-D: | 1 |
| ADD Duration: | 1:53 |
| MEAN Duration: | 1:49 |
| TALK Duration: | 0:14 |
| Links within zone: | 1:2; 1-1; 2:3; 1:2; 1-1 |
| Loners: | 1 |
| Loner duration: | 0:05 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 0 |
| TALK-D end: | 1 |
| TALK-C-SE, C-OE: | 0 |

**Strategies:**

The whole sequence starts as a short introductory talk in which the participant announces exactly what he is about to draw on the whiteboard. With '*skipping some various components in between*' the participant develops the main structure, and then the components of one basic part of the infrastructure (RSA). The participant then directs focus on the main structure, and towards the practical products it delivers. The participant then explains what the actual practice is in the working team – to deliver that transform. Then, the participant develops the transformation structure, with elements in it. The participant

describes the structure of transforms, and adds the transform applyGUI, describing what it is and what it does. The participant finishes with a directional talk – '*So that's the high level overview*'. An illustration of this complete snapshot may be found in Figure 4.18 (Snapshot 3).

**Complete Snapshot (8)**

**Objective measures:**

| | |
|---|---|
| Snapshot Duration: | 5:20 |
| Type of SNAPs preceded | Infra, advinfra |
| # Model Elements: | 12 |
| # corrections: | 0 |
| Pause Duration following Snapshot: | 0:01 |

**Conditions:**

| | |
|---|---|
| ADD: | 15 |
| MEAN: | 11 |
| TALK: | 6 |
| TALK-C-SE: | 1 |
| TALK-C-OE: | 1 |
| TALK-D: | 4 |
| ADD Duration: | 2:31 |
| MEAN Duration: | 1: 58 |
| TALK Duration: | 1-14 |
| Links within zone: | 3:2; 1-1; 1-1; 1-1; 1-1; 3:1; 1-1; 1-1 |
| Loners: | 8 |
| Loner duration: | 1:03 |

**Context:**

| | |
|---|---|
| TALK-D beginning: | 1 |
| TALK-D end: | 1 |
| TALK-C-SE beginning: | 0 |
| TALK-C-SE end: | 1 |
| TALK-C-OE: | 0 |

**Strategies:**

The participant starts with a short directional talk ('*the first thing that everybody does now, when they look at Aurora. After our last release, we agreed on certain high level architectural constructs; so, I'll draw them in UML-type form...*'), provides the layers in the core structure, and finishes at one moment with a concluding talk. Then, he proceeded to offer a complete explanation of mechanics between layers (0-4) in the model ('*The idea is that this sort of layering approach is beneficial because we can distinguish application*

*from platform and as you go up the layers you're increasing your functionality, you start at the layer 0 and you pretty much have just mini tech that are built on top of Eclipse.*').

# Appendix C: Wide Pattern Base (Initial analytic steps)

In this appendix, we introduce a high-level description of a pattern language for software comprehension. Our intent is not to present a final, definitive pattern language. In fact, the patterns community believes that patterns evolve over time; they contain mistakes and it is worthwhile trying to fine-tune them over time. Our intent is, rather, to demonstrate the kinds of things a pattern language for software comprehension might contain, and to set the scene for further development of the language, and, based on the language, for tool support. The patterns we outline were found in the initial analysis of the research described in Chapter 3.

We leave it as an open research question whether these patterns and pattern languages constitute a strong basis for software comprehension. We will, however, demonstrate by example that some parts of the language are grounded in reality.

**Pattern Language Summary**

Figure C1 illustrates the relationships among the highest-level patterns. Arrows point to other patterns which help resolve the forces introduced or partially resolved by a pattern. The patterns in this language jointly resolve difficulties users have in understanding software.

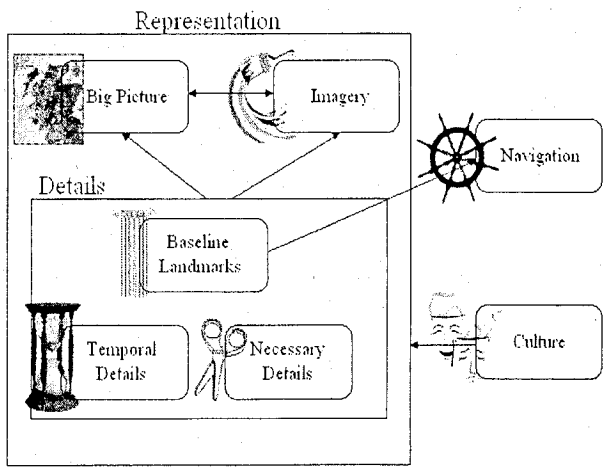


Figure C1: Pattern Language Thumbnail

### The Patterns

Below are concise descriptions of the high-level patterns in our pattern language for software comprehension. We will not go into detail regarding the individual sub-patterns (presented in italics), leaving this as future work. The exception to this is *Temporal Details*, which is the topic of this dissertation.

### Big Picture

Today's monolithic software systems defy human comprehension when presented in their barest form, source code. Furthermore, even understanding abstract representations of large systems, such as UML diagrams, borders on the unmanageable [7]. There is no doubt that the creation of a high-level abstraction may ease the task in understanding. However, human capacity for seeing the "big picture" is poorly understood [1]. This seminal issue serves as a primary motivator for research in reverse engineering [30] software visualization [135] and software architecture [18].

*Big Picture* is central in our pattern language for software comprehension. It breaks down into sub-patterns describing user representation strategies, such as:

- *Set Boundaries*
- Assessing *Level Of Abstraction* [18] (the ability to filter out fine-detail to get the big picture can be an advantage [108])
- Amalgamating *Multiple Views* [110]
- *Incremental Construction* of the big picture

Relationships to other patterns include:

- Show appropriate *Necessary Details*
- Making use of *Culture* (which is termed *Corporate Memory* in [95])

Collectively, the sub-patterns of *Big Picture* address the issue of how to utilize software knowledge at a high level of abstraction.

### Baseline Landmark

During our videotaped whiteboard sessions, participants indicated "I know about this [software component] only in terms of something else", or "I know about this only because

I believe it is similar to another concept I already understand." These quotes are ideal illustrations of the *Baseline Landmark* pattern.

We make use of our previous experiences when deliberating over new maintenance tasks. Moreover, in these prior experiences we encounter entrenched landmarks, or 'points of crossing' with other experiences, which enrich our understanding and become foundations for future comprehension. We return to them regularly in order to understand disparate aspects of the software system. The *Baseline Landmark* is central to the understanding of complex systems, and allows us to understand a software system without starting afresh. Alternatively, this pattern could have been named *'Starting Point'*; however, *'Baseline Landmark'* evokes the notion we return to the landmarks often, and not necessarily as a starting point. *Baseline Landmark* is similar to concepts such as beacons [38, 148], frame-of-reference, and analogical reasoning.

This pattern addresses how users solidify understanding around central details in a variety of topics including:

- *Familiarity* and *Recognition*
- *Centrality*
- *Navigation Links*
- *Reference Points*
- *Non-Central Landmarks*

*Baseline Landmark* may be related to how people navigate complex terrains (for a comparison of program comprehension with urban exploration, see also [94, pp.6-9]). People find some highly recognizable landmark (e.g. a tall building, a river). This landmark seems to tie to many other things and therefore serves as a reference and forms a basis for navigation.

This pattern describes the central details of the greatest importance. To complement this, the next pattern comprises the important visual details in an appropriate representation.

### Imagery

To boldly repeat an overused aphorism, "a picture is worth a thousand different words to a thousand different people." However, the question of which words are worth representing in pictures provokes a number of complex issues. The complexity lies not just in the information to be visualised, but also in the information's context of use [106].

The *Imagery* pattern describes ways by which a software developer represents their mental model of complex software through diagrams and symbols in order to facilitate understanding. This pattern explores further issues of representation, such as:

- Understanding *Static Representations*
- Understanding *Dynamic Behaviour*
- *Changing Perspectives*
- Illustrating *Representations Over Time*
- *Personal Representation* vs. *Cultural Dependency*
- *Controlling Representation*
- *Convention*
- *External Memory*
- *Cognitive Structures*

Through exploration of cognitive procedures for the interpretation of graphic visualizations of software, the *Imagery* pattern explores which diagrammatic techniques correspond to images in the user's mind. The *Big Picture* and *Imagery* patterns together describe the way in which people create selective representations when they understand software. These selective representations may manifest in diagrams, whiteboard sessions, and the backs of napkins, or else may remain merely in the user's mind.

Within a representation, a user accesses and interprets details and landmarks. The next three patterns highlight particular kinds of details people tend to think about and represent when understanding software.

### Necessary Details

Users may understand software better through abstractions they create themselves [129]. Although the reverse engineering and software architecture literature indicates

'create and support abstractions', little empirical validation has been performed to determine which abstractions are valid and appropriate and for which users. The *Necessary Details* pattern explores how users understand what is necessary, but not more than is needed.

When users establish a *Big Picture*, they add and remove details from their representation, and set boundaries outside of which lay unimportant details. Within these boundaries lies *Necessary Details*. *Necessary Details* may change over time, and thus we see a special relationship with *Temporal Details*. This pattern recognizes challenges in comprehending suitability through the following issues:

- *Required Depth* vs. *Inappropriate Depth*
- *Temporal Quality*
- *Big Picture Boundaries*
- How human minds *Add Detail* or *Remove Detail*
- *User Preferences* for necessity

The *Necessary Details* pattern addresses creating and supporting the right abstractions for representations. The next language expresses the flexible evolution of representation within a user's mind.

**Temporal Details**

Everything about our understanding of software is affected by time. Mental representations and their internal details change over time. This pattern illustrates the dynamics of time within the user's mind, and is the topic of this dissertation.

Transitions are the centre of discussion in the next pattern. The *Baseline Landmark*, *Necessary Details*, and *Temporal Details* patterns collectively concentrate on the details of a representation insofar as centrality, importance, and time are concerned. However, when a user accesses a representation, they do not merely add and remove elements over time, creating a gargantuan, unruly representation. Rather, they express their representation in terms of multiple views, and navigate between these views to allow ease of transition through comprehension procedures.

### Navigation

Software engineers perform 'Just in Time Comprehension' [125] by repeatedly searching for source code artefacts and navigating through their relationships. This navigation is broken down into two distinct categories [124], browsing (exploratory, no distinct goal) and searching (planned, distinct goal in mind). The *Navigation* pattern describes how users cognitively browse and search mental representations.

*Navigation* is a principal pattern for software comprehension. Humans navigate to better understand the boundaries of a *Big Picture*, or to assess which are the *Necessary Details*. We navigate from a *Baseline Landmark* to other distinct parts of a system. And finally, how and what we navigate changes over time (*Temporal Details*). Undeniably, navigation is a fundamental human cognitive procedure. This pattern language analyses:

- Strategies for system navigation (i.e. *Use Cases*, *Sequentially*, *Breadth-first*, *Depth-first*, *Opportunistic*, *Polymorphic Implementation*, *Categorized Methods*, etc.)
- Strategies for comprehension (*Bottom-up*, *Top-Down*, *Opportunistic*, *As-Needed*, *Integrated Metamodel*, etc.)
- Humans using their prior experience for further understanding (*Historical Experience*, a special case of understanding by feature)
- *Points Of Crossing*

Hitherto, we discussed a user's cognitive procedures with respect to representation, details, and navigation. To conclude our pattern language, we delineate that which influences the cognitive procedures in the subtlest of ways, culture.

### Culture

Typically, knowledge about architecture and design tradeoffs, engineering constraints, and the application domain only exists in the minds of the software engineers [18]. The organization aims to capture this knowledge through adequate documentation, mentoring, team building, etc. The goal of this approach is to strengthen the organization's ability to solve highly complex problems through the support of the people that work within the organization. These practices form the basis of *Culture*, our final pattern, and one we believe is a rich area of research. In our opinion, tools often support concepts specific to software engineering, but rarely leverage culture to support concepts specific to a domain

or organization. Sub-patterns include *Cultural Memory*, *Mentoring*, and *Organization Specific*.

# Appendix D: Glossary

**Comprehension**: The act or faculty of understanding, especially of writing or speech [105]. See also *Comprehension, Program* and *Comprehension Process*.

Comprehension in computing is not a novel topic [90], though the landscape has changed significantly from mere programming to the complex process of modern software development in which programming is only one component.

Anderson [10] claims comprehension involves a perceptual stage, followed by a parsing stage where words are transformed into a mental representation, and followed by a utilization stage where the mental representation is used. Bernhardt [20] provides references to the early debate over the question "What is Comprehension?" First, she relates Rumelhart's [115] and others' view: understanding is not a process of breaking complex units of language into simpler ones, but rather, a process of taking multiple units and building them into representations. Next, she relates the view held by Brown and Yule [26]: a proficient reader is one who builds an accurate conceptual representation of written materials. Instead of referring to explicit components in a text, those who comprehend the material refer to their inferences and generalizations about a text's meanings [20]. The critical point is that a reader may build a completely inappropriate model of text meaning without becoming aware of the problem.

The interpretation of text is not dissimilar to categorization – chunks of text are combined in memory into larger cognitive units [16]. Like categorization, combining chunks into cognitive units may support the "comprehender" [113]. This support may include *economy* and *communication* (if culturally-shared structures are used to represent texts, interpretations can be shared with those who share those structures).

Textual comprehension is therefore a process involving the building of representations from inferences and generalizations of the meaning of text.

**Comprehension, Program**: (Section 2.4) The process of the acquisition of knowledge about a computer program [114]. *Also called* program understanding or software comprehension[34] [22]

We extend the definition of textual comprehension to software as follows: software comprehension is the process of building representations (e.g. tool output, mental representations) from inferences and generalizations of the meaning of software artefacts (e.g. design documents, source code, test cases) or the communication among software professionals (e.g. in meetings, whiteboard sessions, emails, and off-the-cuff discussions).

The goal of the program comprehension community is to enhance scientific knowledge about understanding, with the ultimate objective of developing more effective tools and methods [128]. We consider program comprehension a challenging manual task, relevant to common software tasks such as software evolution, maintenance and documentation. Many researchers ground their research in program comprehension with the following introductory argument: programmers who engage in software evolution devote more than half of their time to program understanding[35].

Research in program comprehension includes research into such topics as beacons [25, 54, 145], chunking [38, 148], comprehension processes, comprehension models, comprehension strategies (including programming plans), concept assignment [21], the role of expertise [19, 136, 146] (including programming knowledge), mental models, program slicing [144], representation form effects [35], and software psychology. Many of these topics still receive attention. Additionally, new areas of interest include cognitive support [140-142] and individual differences [47, 71].

**Comprehension Process:** (Section 2.4) The process of creating a mental model. A collection of processes, or the study thereof, may contribute to a more complete model of comprehension.

**Diagram**: A two-dimensional symbolic representation, of processes, features, etc. that employs lines, shapes and symbols. See also, *Model.*

[34] The term "software comprehension" is more circumspect, reflecting the nature of modern software development as more significant than programming. The term "program comprehension" is widely used, however. Also the words 'understanding' and 'comprehension' are synonyms.

[35] Modern empirical evidence, confirming this widely accepted claim, would be most welcome.

**Diagrammatic Reasoning**: (Section 2.5) The study of representations; in particular, how natural and artificial agents create, manipulate, reason about, solve problems with, and in general use such representations [11]. Research in diagrammatic reasoning has two goals, beyond understanding the diagramming phenomena and its processes [56, pp.xxi]. The first goal is to deepen our understanding of the way in which we think. The second goal is to provide the scientific base for constructing diagrammatic representations that software can store and manipulate. These goals are not only of interest to design theorists, who regularly contribute to HCI and are absorbed in the role of sketches and diagrams as design aids [56, pp.xv], but also philosophers, cognitive psychologists, logicians, AI researchers, and of course, tool developers.

**Manipulate**: (Chapter 6) Changing a network of recorded snapshots using some form of tool. In particular we consider it possible to manipulate prior versions, that is, the after-the-fact exploration of "what if" scenarios.

**Meaning**: (Chapter 6) The *rationale* for transitions between versions (c.f. versions and transitions).

**Model**: An integrated representation, containing multiple diagrams, each acting as a view of some of the information in the model. See also, Diagram, Model, mental, Model, final and Model, prior.

When discussing a representation of software, we will use the term "model" as opposed to "diagram". Diagrams are found in models, but a model is more: The model as a whole will contain information from many dimensions including form, time, and rationale; not all of the information will appear on diagrams.

**Model, prior:** (Chapter 6) A model version as it existed in the past.

**Model, final:** (Chapter 6) A model as it stands at the present moment, even though the model may evolve still further in the future.

**Model, mental (mental model)**: An internal representation [32, 67, 68] of a situation constructed within an individual's mind in order to manipulate or predict an outcome[36]. See also Comprehension Process.

Mental model research has roots in reading comprehension. Regarding software, a mental model is a mental representation of a human's knowledge of software. Mental model research constitutes the philosophical underpinning of many fields within cognitive science.

Various terms have been used to describe the notion of mental models [126]. We adopt Norman's notion of mental model [100], and Farooq and Dominick's notion of cognitive models [49], which we describe presently. Norman suggests five models of a software system:

- the target system (the software system, source code)
- the conceptual model of the target system (target system representations, e.g. UML models)
- the system image (impression or presentation of software system to users, e.g. software user interface)
- the user's mental model of the target system ("what people really have in their heads and what guides their use of things" [100, pp.12])
- the scientist's conceptualization of the mental model (a model of the mental model)

Farooq and Dominick [49] suggest the fifth point is better described as a "cognitive model." A cognitive model describes the mental processes and information structures humans use to complete a task; to be precise, a researcher's conception of a mental model.

Norman suggests that the root of a system's design or implementation is the designer's mental model [101]. To align the designer's mental model with the user's mental model is deemed essential. The reasoning is that the designer can only communicate with the user through the "system image", the designer's materialised mental model.

There are two implications for tool design. The first inference is the following: tools that aid a software designer should produce the right mental model or the resulting software

[36] Though an open research question, we drive this research from the assumption that people indeed create mental models.

system may suffer. From this inference we deduce that tool developers should anticipate the mental models produced by their tools. The second inference is critical: tool developers will inevitably design their tools based on their own mental model. Based on these inferences, we arrive at a critical point. Tool developers must have the right mental model for themselves and for their users[37].

**Pattern:** (Section 2.7 and Chapter 6) A structured exposition of a solution to a recurring problem in a specific context through which readers gain understanding.

Although this is our definition of pattern, the definition of pattern is a source of debate [138]. A common definition, from Alexandrian theory, is that a pattern is a solution to a problem in a context. This definition describes elements of a pattern, but has been dismissed (most notably in [137]) on account of lacking notions of recurrence, teaching, naming and structure.

According to Lea [76], our definition will not suffice. If we want to follow Alexander and produce quality to make people feel more alive, then patterns must also include properties of encapsulation, generativity, equilibrium, abstraction, openness, and composability. However, as with anything that needs defining, the definition cannot include every aspect.

Each pattern encapsulates a well-defined solution to a problem such that we know the problem is real, and when to apply the pattern. Generativity, despite the confusing discussion in the literature, simply means a pattern helps the reader solve problems the pattern does not address explicitly, i.e. *emergent* problems. Equilibrium signifies that the intricacies and variables influencing a problem (known as forces) come into balance through the solution[38]. Patterns represent abstractions of empirical experience in addition to common knowledge. Openness means patterns may be extended to arbitrarily fine levels of detail. We can sub-divide a pattern into many smaller patterns. Composability, finally, refers to pattern languages expressing the layering between higher and lower-level patterns.

[37] In this context, technically speaking, the users' mental model is the cognitive model.

[38] A classic example is that efficiency and maintainability might be balanced. In the earliest days, efficiency (time) and memory requirements (space) were traditional forces in software development.

**Pattern Language:** (Section 2.7 and Chapter 6) A *pattern language* is a high-level pattern that breaks down into lower-level patterns or pattern languages, thus forming a hierarchy. Pattern names collectively form a vocabulary, which facilitates communication between people within a particular field.

**System**: (Chapter 6) The software being modelled. Chapter 6 presents patterns for teaching or understanding an existing complex systems through the details of the system history, such as prior states or decisions.

**Theory:** (Sections 2.6 and 5.3) A set of interrelated constructs (variables), definitions, and propositions that presents a systematic view of phenomena by specifying relations among variables with the purpose of explaining natural phenomena [70, pp.64]. The notion of interrelated constructs also contains the idea of theoretical rationale, "specifying how and why the variables and relational statements are interrelated" [74, pp.17]. The theory relates independent and dependent variables in a study, and then provides an explanation for how and why the code explains or predicts the dependent variable. One way to develop theories is to build them through field observation.

The notion of theory is essential to science and engineering. Scientists develop and validate theories; engineers apply validated theories. According to Newell and Card [98], "Nothing drives basic science better than a good applied problem." Likewise, Lewin [82] states, "There is nothing so useful as a good theory." However, in the same vein, the theorist should heed a popular quote of Friedrich Engels, "An ounce of action is worth a ton of theory."

**Transition**: (Chapter 6) A change between versions. See Version.

**Version**: (Chapter 6) The state of a model after a group of changes. We are interested in the versions of models under development as well as versions of models representing discrete software releases.

A recurring theme in the Temporal Details chapter is the model that evolves over time. Models may evolve over time because they are developed over time, or because the artefact the model represents changes over time, and therefore the model correspondingly adjusts to match or capture this change. See Transition, Model.

References

[1] Large Scientific and Software Data Set Visualization. National Science Foundation, Program Announcement NSF-99-105, 1999.

[2] *Proc.* 11th Working Conference on Reverse Engineering (WCRE'04), Delft, Netherlands, November 8-12, 2004.

[3] UML™ Resource Page, in www.uml.org, Object Management Group (OMG).

[4] Model-Driven Architecture: A Technical Perspective, in ftp://ftp.omg.org/pub/docs/ab/01-02-01.pdf, OMG Architecture Board MDA Drafting Team.

[5] *Proc.* 12th Pattern Languages of Programs (PLoP 2005), Champaign, USA, September 2005.

[6] Ader, H., Roeline, H., Pasman, W., and The, B. Generating New Research Hypotheses from a Result Diagram of Qualitative Research. In A. Blackwell, K. Marriott, and A. Shimojima, Eds, *Diagrammatic Representation and Inference, Lecture Notes in Artificial Intelligence (LNAI) 2980.* Berlin-Heidelberg: Springer-Verlag, 2004, pp. 329-332.

[7] Aldrich, J. *Using Types to Enforce Architectural Structure.* PhD Thesis, Computer Science and Engineering, University of Washington, 2003.

[8] Alexander, C. *The Timeless Way of Building.* New York, NY: Oxford University Press, 1979.

[9] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. *A Pattern Language.* New York, NY: Oxford University Press, 1977.

[10] Anderson, J. R. *Cognitive Psychology and Its Implications*, 5th ed. ed: Worth Publishers, 1999.

[11] Anderson, M. Diagrammatic Reasoning Site. http://www.cs.hartford.edu/~anderson/, accessed April 10, 2006.

[12] Appleton, B., Berczuk, S., Cabrera, R., and Orenstein, R. Streamed Lines: Branching Patterns for Parallel Software Development. *Proc.* Pattern Languages of Programs (PLoP '98), Monticello, Illinois, USA, 1998.

[13] Arunachalam, V. and Sasso, W. Cognitive processes in program comprehension: an empirical analysis in the context of software reengineering. *Systems and Software*, vol. 34:177-189, 1996.

[14] Backman, K. Challenges of the grounded theory approach to a novice researcher. *Nursing and Health Sciences*, vol. 1:147-153, 1999.

[15] Barnard, P. and May, J. Cognitive Modelling for User Requirements. In P. Byerley, P. Barnard, and J. May, Eds, *Computers, Communication and Usability: Design issues, research and methods for integrated services*. Amsterdam: Elsevier, 1993, pp. 101-146.

[16] Barsalou, L. and Sewell, D. Contrasting the representation of scripts and categories. *J. of Memory and Language*, vol. 24:646-665, 1985.

[17] Basili, V. R. The Role of Experimentation in Software Engineering: Past, Current, and Future. *Proc.* 18th Int. Conf. on Software Engineering (ICSE), Los Alamitos, CA, pp. 442-450, IEEE Computer Society Press, 1996.

[18] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 1998.

[19] Berlin, L. M. Beyond Program Understanding: A Look at Programming Expertise in Industry. *Proc.* Empirical Studies of Programmers: 5th Workshop (ESP'93), pp. 8-25, 1993.

[20] Bernhardt, E. Proficient Texts or Proficient Readers. *ADFL (Association of Departments of Foreign Languages) Bulletin*, vol. 18:25-28, 1986.

[21] Biggerstaff, T., Mitbander, B., and Webster, D. The concept assignment problem in program understanding. *Proc.* Int. Conf. on Software Engineering (ICSE'93), Los Alamitos, CA, pp. 482-498, IEEE Computer Society Press, 1993.

[22] Boehm-Davis, D. Software Comprehension. In M. Helander, Ed. *Handbook of Human-Computer Interaction*: Elsevier Science Publishers, 1988.

[23] Bogdan, R. and Bilken, S. *Qualitative research for education: An introduction to theory and methods*. Boston: Allyn and Bacon, 1992.

[24] Brooks, R. Studying Programmer Behavior Experimentally: The Problems of Proper Methodology. *CACM*, vol. 23:207-213, 1980.

[25] Brooks, R. Towards a theory of the comprehension of computer programs. *Int. J. of Man-Machine Studies*, vol. 18:543-554, 1983.

[26] Brown, G. and Yule, G. *Discourse Analysis*. Cambridge, UK: Cambridge United Press, 1984.

[27] Butcher, K. and Kintsch, W. Learning with Diagrams: Effects on Inferences and the Integration of Information. In A. Blackwell, K. Marriott, and A. Shimojima, Eds,

*Diagrammatic Representation and Inference, Lecture Notes in Artificial Intelligence (LNAI) 2980*. Berlin-Heidelberg: Springer-Verlag, 2004, pp. 337-340.

[28] Carver, J. *The Impact of Background and Experience on Software Inspections*. Ph.D. Thesis, Computer Science, University of Maryland, College Park, MD, 2003.

[29] Cheng, P. Why Diagrams Are (Sometimes) Six Times Easier than Words: Benefits beyond Locational Indexing. In A. Blackwell, K. Marriott, and A. Shimojima, Eds, *Diagrammatic Representation and Inference, Lecture Notes in Artificial Intelligence (LNAI) 2980*. Berlin-Heidelberg: Springer-Verlag, 2004, pp. 242-260.

[30] Chikofsky, E. J. and Cross II, J. Reverse Engineering and Design Recovery: A Taxonomy, in *IEEE Software*, vol. 7, pp. 13-17, 1990.

[31] Coplien, J. The Column without a Name: The Human Side of Patterns, in *C++ Report*, vol. 8, pp. 81, 1996.

[32] Craik, K. *The Nature of Exploration*. Cambridge, UK: Cambridge University Press, 1943.

[33] Creswell, J. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, 2nd ed. Thousand Oaks, California: Sage Publications, 2003.

[34] Cross II, J., Hendrix, T., Barowski, L., and Mathias, K. Scalable visualizations to support reverse engineering: A framework for evaluation. *Proc.* 5th Working Conference on Reverse Engineering (WCRE'98), Honolulu, Hawaii, USA, pp. 201-209, IEEE Computer Society Press, 1998.

[35] Cunniff, C. and Taylor, C. Representation form effects on novice's program comprehension. *Proc.* 2nd Workshop Empirical Studies of Programmers (ESP'87), 1987.

[36] Curtis, B. Measurement and Experimentation in Software Engineering. *IEEE*, vol. 68:1144-1157, 1980.

[37] Curtis, B. By the way, did anyone study any real programmers? *Proc.* Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86), pp. 256-262, 1986.

[38] Davis, J. S. Chunks: A Basis for Complexity Measurement. *Information Processing & Management*, vol. 20:119-127, 1984.

[39] Demeyer, S., Ducasse, S., and Nierstrasz, O. *Object-Oriented Reengineering Patterns*: Morgan Kaufmann and DPunkt, 2002.

[40] Diskin, Z. Visualization vs. Specification in Diagrammatic Notations: A Case Study with the UML. In M. Hegarty, B. Meyer, and N. Narayanan, Eds, *Diagrammatic Representation and Inference, Lecture Notes in Artificial Intelligence (LNAI) 2317.* Berlin-Heidelberg: Springer-Verlag, 2002, pp. 112-115.

[41] Dreyfus, S. Formal Models vs. Human Situational Understanding: Inherent Limitations on the Modeling of Business Expertise. *Office: Technology and People*, vol. 1:133-165, 1982.

[42] D'Souza, D. Model-Driven Architecture and Integration: Opportunities and Challenges, in *http://www.catalysis.org/publications/papers/2001-mda-reqs-desmond-6.pdf.*

[43] Ducasse, S., Girba, T., and J.-M., F. Modeling Software Evolution by Treating History as a First Class Entity. *Proc.* Workshop on Software Evolution Through Transformation (SETra 2004), pp. 71-82, 2004.

[44] Eades, P., Lai, W., Misue, K., and Sugiyama, K. Preserving the mental map of a diagram. *Proc.* Compugraphics 91, pp. 24-33, Springer LNCS/AI, 1991.

[45] Erickson, F. Some approaches to inquiry in school-community ethnography. *Anthropology and Education Quarterly*, vol. 8:58-69, 1977.

[46] Ericsson, K. A. and Simon, H. A. *Protocol Analysis: Verbal Reports As Data.* Cambridge, MA: MIT Press, 1984.

[47] Exton, C. Constructivism and Program Comprehension Strategies. *Proc.* 10th Int. Workshop on Program Comprehension (IWPC), Paris, France, pp. 281-284, IEEE Computer Society Press, 2002.

[48] Farah, H. *Applying cognitive patterns to support software tool development.* (M.Sc. Draft), Computer Science, University of Ottawa, Ottawa, 2006.

[49] Farooq, M. and Dominick, W. A survey of formal tools and models for developing user interfaces. *Int. J. of Man-Machine Studies*, vol. 29:479-496, 1988.

[50] Feynman, R. *"Surely you're joking Mr. Feynman!": Adventures of a curious character.* New York: W.W.Norton & Co., 1985.

[51] Freedman, D. *Corps Business: The 30 Management Principles of the U.S. Marines.* London: Collins, 2001.

[52] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley Publishing Company, 1995.

[53] Gardner, K., Rush, A., Konitzer, B., Crist, M., and Teegarden, B. *Cognitive Patterns: Problem Solving Frameworks for Object Technology*. Cambridge, UK: Cambridge University Press, 1998.

[54] Gellenbeck, E. and Cook, C. An investigation of procedure and variable names as beacons during program comprehension. *Proc.* Empirical Studies of Programmers: 4th Workshop (ESP'91), pp. 65-81, 1991.

[55] Glaser, B. and Strauss, A. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Chicago, USA: Aldine, 1967.

[56] Glasgow, J., Narayanan, N., and Chandrasekaran, B. *Diagrammatic Reasoning: Cognitive and Computational Perspectives*. Cambridge, MA and London, UK: AAAI Press | The MIT Press, 1995.

[57] Green, T. Programming as a cognitive activity. In H. Smith and T. Green, Eds, *Human Interaction With Computers*. New York: Academic Press, 1980.

[58] Hahn, J. and Kim, J. Why are some diagrams easier to work with? Effects of Diagrammatic Representation on the Cognitive Integration Process of Systems Analysis and Design. *ACM Transactions on Computer Human Interaction*, vol. 6:181-213, 1999.

[59] Harper, D. Visual sociology: Expanding sociological vision. In J. Grant and E. Brent, Eds, *New technology in sociology: Practical applications in research and work*. New Brunswick, NJ: Transaction Publishers, 1989, pp. 81-97.

[60] Harrison, N. The Language of Shepherding: A Pattern Language for Shepherds and Sheep. *Proc.* 7th Pattern Languages of Programs Conference (PLoP), Illinois, USA, 2000.

[61] Herbsleb, J., Klein, H., Olson, G., Brunner, H., Olson, J., and Harding, J. Object-oriented analysis and design in software project teams. *Human Computer Interaction*, vol. 10:249-292, 1995.

[62] Herbsleb, J. and Kuwana, E. Preserving knowledge in software engineering: What designers need to know. *Proc.* INTERCHI'93, New York, pp. 7-14, ACM, 1993.

[63] Herrera, F. *A Usability Study of the "TkSee" Software Exploration Tool*. M.Sc., Computer Science, University of Ottawa, Ottawa, 1999.

[64] http://www.aaai.org/AITopics/html/casebased.html. Case-Based Reasoning (a sub-topic of Reasoning), Accessed Sept 26, 2006.

[65] http://www.omg.org/mda/. OMG Model-Driven Architecture Home Page:, 2005.

[66] Jarczyk, A., Loffler, P., and Shipman, F. Design Rationale for Software Engineering: A Survey. *Proc.* Int. Conf. on System Sciences, Hawaii, USA, pp. 577-586, IEEE Computer Society Press, 1992.

[67] Johnson-Laird, P. *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness.* Cambridge, UK: Cambridge University Press, 1983.

[68] Johnson-Laird, P. Mental Models. *Foundations of Cognitive Science*: MIT Press, 1989, pp. 469-499.

[69] Kantrovich, L. To Innovate or Not to Innovate. *Interactions*, vol. 11:25-31, 2004.

[70] Kerlinger, F. *Behavioral research: A conceptual approach.* New York: Holt, Rinehart and Winston, 1979.

[71] Ko, A. J. and Utl, B. Individual Differences in Program Comprehension Strategies in Unfamiliar Programming Systems. *Proc.* 11th IEEE Int. Workshop on Program Comprehension (IWPC), IEEE Computer Society Press, 2003.

[72] Kotov, A. *Think Like a Grandmaster.* London: Batsford Chess Books, 1971.

[73] Kunz, W. and Rittel, H. Issues as Elements of Information Systems. Working Paper No 131, University of California, Berkeley 1970.

[74] Labovitz, S. and Hagedorn, R. *Introduction to social research.* New York: McGraw-Hill, 1971.

[75] Larkin, J. and Simon, H. A. Why a Diagram is (Sometimes) Worth Ten Thousand Words. In J. Glasgow, N. Narayanan, and B. Chandrasekaran, Eds, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*. Cambridge, MA and London, UK: AAAI Press | The MIT Press, 1995, pp. 69-109.

[76] Lea, D. Christopher Alexander: An Introduction For Object-Oriented Designers, in *Software Engineering Notes*, vol. 19, pp. 39-45, 1993.

[77] Leffingwell, D. and Widrig, D. *Managing Software Requirements: A Unified Approach.* Reading, MA: Addison-Wesley, 2000.

[78] Lethbridge, T. Integrated Personal Work Management in the TkSee Software Exploration Tool. *Proc.* Second International Symposium on Constructing Software Engineering Tools (CoSET2000), in association with ICSE 2000, Limerick, Ireland, pp. 31-38, 2000.

[79] Lethbridge, T., Sim, S., and Singer, J. Studying Software Engineers: Data Collection Methods for Software Field Studies. *Empirical Software Engineering*, to appear, 2004.

[80] Letovsky, S. Cognitive processes in program comprehension. *Proc.* Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86), pp. 58-79, 1986.

[81] Letovsky, S., Pinto, J., Lampert, R., and Soloway, E. A cognitive analysis of code inspection. *Proc.* Empirical Studies of Programmers: Second Workshop, pp. 231-247, 1989.

[82] Lewin, K. *Field theory in social science.* New York: Harper Row, 1951.

[83] Lewis, C. Using the "Thinking-Aloud" method in Cognitive Interface Design. IBM Thomas J. Watson Research Center, Yorktown Heights, New York, Research Report 1982.

[84] Lewis, C. and Olson, G. Can principles of cognition lower the barriers to programming? *Proc.* Second Workshop on Empirical Studies of Programmers, Washington, DC, 1987.

[85] Littman, D., Pinto, J., Letovsky, S., and Soloway, E. Mental models and software maintenance. *Proc.* Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86), pp. 80-98, 1986.

[86] Locke, L., Spirduso, W., and Silverman, S. *Proposals that work: A guide for planning dissertations and grant proposals*, 4th ed. Thousand Oaks, CA: Sage Publications, 2000.

[87] Manns, M. L. and Rising, L. *Fearless Change: Patterns for Introducing New Ideas.* Boston, MA: Addison-Wesley Professional, 2004.

[88] Martin, D., Rodden, T., Rouncefield, M., Sommerville, I., and Viller, S. Finding Patterns in the Fieldwork. *Proc.* 7th European Conf. on Computer Supported Cooperative Work, Bonn, Germany, pp. 39-58, Kluwer Academic Publishers, 2001.

[89] Martin, R., Buschmann, F., and Riehle, D. *Pattern Languages of Program Design 3 (PLoPD3).* Reading, MA: Addison-Wesley, 1997.

[90] Mayer, R., Dyck, J., and Vilberg, W. Learning to Program and Learning to Think: What's the Connection? *CACM*, vol. 29:605-610, 1986.

[91] McCall, R. J. PHI: A Conceptual Foundation for Design Hypermedia. *Design Studies*, vol. 12:30-41, 1991.

[92] Meszaros, G. and Doble, J. A Pattern Language for Pattern Writing. *Proc.* Conference on Pattern Languages of Program Design (PLoP '98), pp. 529-574, 1998.

[93] Miles, M. and Huberman, A. *Qualitative Data Analysis*, 2nd ed. Thousand Oaks, California: SAGE Publications, 1994.

[94] Moonen, L. *Exploring Software Systems*. Ph.D., Faculty of Natural Sciences, Mathematics and Computer Science, University of Amsterdam, 2002.

[95] Muller, H. A., Jahnke, J., Smith, D., Storey, M.-A. D., Tilley, S., and Wong, K. Reverse Engineering: a roadmap. In A. Finkelstein, Ed. *The Future of Software Engineering*: ACM Press, 2000.

[96] Murray, A. and Lethbridge, T. Cognitive Patterns for Software Comprehension: Temporal Details. *Proc.* 12th Pattern Languages of Programs (PLoP 2005), Champaign, USA, 2005.

[97] Narayanan, N. Reasoning with Diagrammatic Representations. AAAI Press, Menlo Park, CA SS-92-02, 1992.

[98] Newell, A. and Card, S. K. The prospects for psychological science in human-computer interaction. *Human Computer Interaction*, vol. 1:209-242, 1985.

[99] Nielsen, J. *Usability Engineering*. San Francisco, CA: Morgan Kaufmann, 1993.

[100] Norman, D. Some Observations on Mental Models. In D. Genter and A. Stevens, Eds, *Mental Models*. Hillsdale, NJ: Lawrence, Erlbaum Assoc., 1983, pp. 15-34.

[101] Norman, D. *The Design of Everyday Things*. New York: Doubleday, 1988.

[102] Nunan, D. *Research methods in language learning*. Cambridge, UK: Cambridge University Press, 1992.

[103] O'Brien, M., Buckley, J., and Exton, C. Empirically Studying Software Practitioners - Bridging the Gap between Theory and Practice. *Proc.* Int. Conf. on Software Maintenance (ICSM), Budapest, Hungary, pp. 433-442, IEEE Computer Society Press, 2005.

[104] Olson, G., Olson, J., Carter, M., and Storrosten, M. Small group design meetings: An analysis of collaboration. *Human Computer Interaction*, vol. 7:347-374, 1992.

[105] Pearsall, J. *The Concise Oxford Dictionary*, 10th edition ed: Oxford University Press, 1999.

[106] Petre, M., Blackwell, A., and Green, T. Cognitive Questions in Software Visualization. *Software Visualization: Programming as a Multi-Media Experience*: MIT Press, 1997, pp. 453-480.

[107] Pidgeon, N., Turner, B., and Blockley, D. The Use of Grounded Theory for Conceptual Analysis in Knowledge Elicitation. *Int. J. of Man-Machine Studies*, vol. 35:151-173, 1991.

[108] Price, B. A., Baecker, R. M., and Small, I. S. A principled taxonomy of software visualization. *Visual Languages and Computing*, vol. 4:211-266, 1993.

[109] Qin, Y. and Simon, H. A. Imagery and Mental Models. In J. Glasgow, N. Narayanan, and B. Chandrasekaran, Eds, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*. Cambridge, MA and London, UK: AAAI Press | The MIT Press, 1995, pp. 403-434.

[110] Ramasubbu, S. *Reverse Software Engineering Large Object Oriented Software Systems using the UML Notation*. M.Sc., Elec. Eng., Virginia Polytechnic Institute and State University, Virginia, 2001.

[111] Rising, L. and Coplien, J. *The Pattern Handbook: Techniques, strategies and applications*: SIGS Reference Library, 1998.

[112] Robillard, P., d'Astous, P., Détienne, F., and Visser, W. Measuring Cognitive Activities in Software Engineering. *Proc.* 20th Int. Conf. on Software Engineering (ICSE), Kyoto, Japan, pp. 292-301, IEEE Computer Society Press, 1998.

[113] Rosch, E. Principles of Categorization. In E. Rosch and B. Lloyd, Eds, *Cognition and Categorization*. Hillsdale, NJ: Lawrence Erlbaum Assoc, 1978, pp. 27-48.

[114] Rugaber, S. Program Understanding. In A. Kent and J. G. Williams, Eds, *Encyclopaedia of Computer Science and Technology*, 1996, pp. 341-368.

[115] Rumelhart, D. *Introduction to Human Information Processing*. New York: Wiley, 1977.

[116] Russo, J., Johnson, E., and Stephens, D. The validity of verbal protocols. *Memory and Cognition*, vol. 17:759-769, 1989.

[117] Schank, R., Kass, A., and Riesbeck, C. *Inside Case-Based Explanation*: Lawrence Erlbaum Associates, 1994.

[118] Seaman, C. *Organizational Issues in Software Development: An Empirical Study of Communication*. PhD Thesis, University of Maryland College Park, 1996.

[119] Seaman, C. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, vol. 25:557-572, July 1999.

[120] Seaman, C. and Basili, V. R. Communication and Organization: An Empirical Study of Discussion in Inspection Meetings. *IEEE Transactions on Software Engineering*, vol. 24:559-572, July, 1998.

[121] Schank, R. Conceptual dependency: A theory of natural language understanding. *Cognitive Psychology*, vol. 3:552-631, 1972.

[122] Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems*: Winthrop Publishers, Inc., 1980.

[123] Sillito, J., De Volder, K., Fisher, B., and Murphy, G. R. Managing Software Change Tasks: An Exploratory Study. *Proc.* Int. Advanced School of Empirical Software Engineering (ISESE 2005), Noosa Heads, Australia, IEEE Computer Society Press, 2005.

[124] Sim, S., Clarke, C., Holt, R., and Cox, A. Browsing and Searching Software Architectures. *Proc.* Int. Conf. on Software Maintenance (ICSM), Oxford, England, pp. 381-390, IEEE Computer Society Press, 1999.

[125] Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. An Examination of Software Engineering Work Practices. *Proc.* CASCON'97, Toronto, ON, Canada, pp. 209-223, 1997.

[126] Staggers, N. and Norcio, N. Mental models: concepts for human-computer interaction research. *International Journal of Man-Machine Studies*, vol. 38:587-605, 1993.

[127] Standish, T. A. An essay on software reuse. *IEEE Transactions on Software Engineering*, vol. 10:494-497, 1984.

[128] Storey, M.-A. D. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. *Proc.* 13th IEEE Int. Workshop on Program Comprehension (IWPC), St.Louis, MO, pp. 181-191, IEEE Computer Society Press, 2005.

[129] Storey, M.-A. D., Fracchia, F. D., and Muller, H. A. Cognitive design elements to support the construction of a mental model during software exploration. *Software Systems (special issue on Program Comprehension)*, vol. 44:171-185, 1999.

[130] Storey, M.-A. D., Muller, H. A., and Wong, K. How do program understanding tools affect how programmers understand programs? *Proc.* Fourth Working Conf. on Reverse Engineering (WCRE), Amsterdam, Netherlands, pp. 12-21, IEEE Computer Society Press, 1997.

[131] Strauss, A. and Corbin, J. *Basics of Qualitative Research. Grounded Theory Procedures and Techniques*. Newbury Park, USA: Stage Publishing Company, 1990.

[132] Suwa, M. and Tversky, B. External Representations Contribute to the Dynamic Construction of Ideas. In M. Hegarty, B. Meyer, and N. Narayanan, Eds, *Diagrammatic Representation and Inference, Lecture Notes in Artificial Intelligence (LNAI) 2317*. Berlin-Heidelberg: Springer-Verlag, 2002, pp. 341-343.

[133] Thizy, D. and Murray, A. On the refactoring of a distributed chess analysis tool. University of Ottawa Computer Science Technical report TR-2002-12, 2002.

[134] Thomas, J. C. A Pattern Language: http://www.truthtable.com/patterns.html, 2004.

[135] Tufte, E. *Envisioning Information*: Graphics Press, 1990.

[136] Vessey, I. Expertise in debugging computer programs: a process analysis. *Int. J. of Man-Machine Studies*, vol. 23:459-494, 1985.

[137] Vlissides, J. Pattern Hatching: Seven Habits of Successful Pattern Writers, in *C++ Report*, vol. 8, 1995.

[138] Vlissides, J. Pattern Hatching: The Top Ten Misconceptions, in *Object Magazine*, vol. 7, pp. 30-33, 1997.

[139] von Mayrhauser, A. and Vans, A. Program comprehension during software maintenance and evolution, in *IEEE Computer*, pp. 44-55, 1995.

[140] Walenstein, A. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD Thesis, Simon Fraser University, 2002.

[141] Walenstein, A. Foundations of Cognitive Support: Toward Abstract Patterns of Usefulness. *Proc.* 14th Annual Conference on Design, 2002.

[142] Walenstein, A. Theory-based Analysis of Cognitive Support in Software Comprehension Tools. *Proc.* 10th Int. Workshop on Program Comprehension (IWPC), pp. 75-84, IEEE Computer Society Press, 2002.

[143] Wales, F. Re: PPIG discuss: Java and Life-Drawing: PPIG Discussion Group, February 6th, 2006.

[144] Weiser, M. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[145] Wiedenbeck, S. Beacons in Computer Program Comprehension. *International Journal of Man-Machine Studies*, vol. 25:697-709, 1986.

[146] Wiedenbeck, S. and Fix, V. Characteristics of the mental representations of novice and expert programmers: an empirical study. *Int. J. of Man-Machine Studies*, vol. 39:795-812, 1993.

[147] Wolcott, H. F. Posturing in qualitative inquiry. In M. D. Le Compte, W. L. Millroy, and J. Preissle, Eds, *The handbook of qualitative research in education*. New York: Academic Press, 1992, pp. 3-52.

[148] Ye, N. and Salvendy, G. Quantitative and Qualitative Differences between experts and novices in chunking computer software knowledge. *Int. J. of Human Computer Interaction*, vol. 6:105-118, 1994.